TotalView

Command Line Interface Guide

August 2001

Version 5.0

Contents

1 TotalView Command Line Interface

What Is the CLI 1
Document Contents 2
Conventions 4
Reporting Problems 4

2 A Few CLI/Tcl Macros

Setting the EXECUTABLE_PATH State Variable 7
Initializing an Array Slice 9
Printing an Array Slice 9
Writing an Array Variable to a File 10
Setting Breakpoints 11

3 Groups, Processes, and Threads

A Couple of Processes 15
Some Threads 17
Even More Complicated Programming Models 19
More on Threads 20
    Types of User Threads 22
Organizing Chaos 23
Creating Groups 28
Simplifying What You're Debugging 35
Setting Process and Thread Focus 38
    Process/Thread Sets 38
    Arenas 39
    GOI, POI, and TOI 40
    Specifying Processes and Threads 41
        The Thread of Interest 41
        Process and Thread Widths 42

Examples 44
Setting Group Focus 44
Specifying Groups in P/T Sets 46
Specifier Combinations 48
All Does Not Always Mean All 50
Setting Groups 52
An Extended Example 54
Incomplete Arena Specifiers 57
Lists with Inconsistent Widths 58
Stepping 59
Thread 59
Process 59
Group 60
Using duntil 60
How do I 61
"Piling Up" vs. "Running Through" 62
P/T Set Expressions 63

4 Using the CLI

How a Debugger Operates 67
Tcl and the CLI 68
The CLI and TotalView 68
The CLI Interface 69
Starting the CLI 70
Initializing the Debugger 71
Start-up Example 72
Starting Your Program 73
CLI Output 76
"more" Processing 77
Command Arguments 77
Symbols 78
Namespaces 78
Symbol Names and Scope 79
Qualifying Symbol Names 80
Command and Prompt Formats 81
Built-In Aliases and Group Aliases 82
Effects of Parallelism on TotalView and CLI Behavior 83
Kinds of IDs 84
Controlling Program Execution 85
Advancing Program Execution 85

Action Points 86

5 Type Transformations

Type Transformation Defined 87
Creating Type Transformations 89
Using Type Transformation 89
Defining Prototypes 91
Objects Used in Type Transformation 94
Creating a struct Type Transformation 95
Validating the Type: the list_validate Procedure 95
Redefining the Type: list_type Procedure 96
Creating the Prototype 97
Making a Callback for a Structure Element 97
Applying Prototypes to Images 98
Initializing TotalView After Loading an Image 99
An Array-Like Example 100
Indicating if a Type Is Mapped: The vector_validate Callback 100
Returning the Type: The vector_type Callback 102
Returning the Rank: The vector_rank Callback 102
Returning the Bounds: The vector_bounds Callback 102
Returning the Address: The vector_address Callback 103
The typedef Callback 104
Utility Procedures 104
The read_store Utility Procedure 105
The ultimate_base Utility Procedure 105
The extract_offset Utility Procedure 106
Creating the Prototype 106
Distributed Arrays 107
Visualizing a Distributed Array with Node Information 109
The Type Transformation for mandel.c 110
Validating the Data Type: The da_validate Function 110
The da_distribution Callback Procedure 112
The Distributed Addressing Callback 113
Addressing Expressions 114
Debugging Tcl Callback Code 118

6 CLI Commands

Command Overview 119
actionpoint 123
alias 126

capture 128
dactions 129
dassign 132
dattach 134
dbarrier 137
dbreak 142
dcheckpoint 145
dcont 148
ddelete 150
ddetach 151
ddisable 152
ddown 153
dec2hex 155
denable 156
dfocus 157
dgo 160
dgroups 162
dhalt 168
dhold 169
dkill 170
dlappend 171
dlist 172
dload 175
dnext 177
dnexti 180
dout 183
dprint 186
dptsets 190
drerun 193
drestart 195
drun 197
dset 200
dstatus 209
dstep 211
dstepi 215
dunhold 217
dunset 218
duntil 219
dup 222
dwait 224

dwatch 225
dwhat 229
dwhere 233
dworker 235
errorCodes 236
exit 238
focus_groups 239
focus_processes 240
focus_threads 241
group 242
help 244
hex2dec 246
image 247
process 250
prototype 253
quit 257
respond 258
stty 259
source_process_startup 260
thread 261
type 263
unalias 266

A CLI Command Summary 267

B CLI Command Default Arena Widths 277

C Distributed Array Type Mapping 281

The cyclic_array.tcl Type Mapping File 286

Glossary 291

Citations 304

Index 305

Chapter 1

TotalView Command Line Interface

This document describes the TotalView® Command Line Interface (CLI). The CLI is a command-oriented debugger that can be used as a stand-alone product or it can be used along with the TotalView Graphical User Interface (GUI) debugger. Depending on your needs, you can view the CLI and TotalView debuggers as being either independent or complementary products. In most cases, the easiest way to debug programs is by using the TotalView GUI. However, there are circumstances when you need to do something not possible or practical using a GUI. For example, you may not want to interactively debug a program that takes days to execute.

The CLI debugger commands that you will execute are integrated within a Tcl interpreter. This combination removes the CLI from the realm of standard command-line debuggers in that you can add your own debugging commands, automate repetitive tasks, and even have the CLI run your program to a point where you are ready to begin debugging with the GUI. For example, you could ask the CLI to watch a memory location for changes and stop the program when a change occurs.

What Is the CLI

The CLI and TotalView are tools that give you visibility into, and control over, your programs. These programs can already be executing or you can load them into memory directly under their control.

The executing program has one or more *processes*, each associated with an executable (and perhaps one or more shared libraries) and each occupying a memory address space. Every process, in turn, has one or more *threads*, each with its own set of registers and its own stack.

The program being debugged is the complete set of threads and communicating processes that make up an application. The exact number of processes and threads depends on many factors, including how you wrote the program, the transformations performed by the compiler, the way the program was invoked, and the sequence of events that occur during execution. Thus, the number of processes and threads usually changes while your program executes.

Some operating systems, compilers, and run-time systems impose restrictions on the relationship between processes, threads, and executables. SPMD (Single Program Multiple Data) programs are parallel programs involving just one executable, executed by multiple threads and processes. MPMD (Multiple Program Multiple Data) programs involve multiple executables, each executed by one or more threads and processes.

Document Contents

Using the CLI assumes that you are familiar with and have experience debugging programs with the TotalView GUI. As CLI commands are embedded within a Tcl interpret, you will get better results if you are familiar with Tcl. However, if you do not know Tcl, you will still be able to use the CLI. What you will loose is the programmability features that Tcl gives. For example, CLI commands operate upon a set of process and threads. You can save this set and apply it to commands based upon what you have saved.

You can obtain information on using Tcl at many book stores, and you can also order these books from online bookstores. Two excellent books are

- Ousterhout, John K. *Tcl and the Tk Toolkit*. Reading, Mass.: Addison Wesley, 1997.
- Welch, Brent B. *Practical Programming in Tcl & Tk*. Upper Saddle River, N.J.: Prentice Hall PTR, 1997.

There is also a rich supply of resources available on the Web. Two starting points are **tcl.activestate.com** and **www.tcltk.com**.

The best way to understand the kinds of information in this book is to take a minute or two to browse through this book's table of contents. The fast-

est way to gain an appreciation of the actions performed by CLI commands is to review Appendix A, which contains an overview of CLI commands.

Here is how the information in this book is organized:

Chapter 1: TotalView Command Line Interface

This first chapter introduces the CLI.

Chapter 2: A Few CLI/Tcl Macros

Because you already know how to program, your biggest challenge in using the CLI will be remembering its commands and understanding how they are used within the Tcl environment. This chapter presents a few macros that demonstrate how the two are used together.

Chapter 3: Groups, Processes, and Threads

Debugging multiprocess, multithreaded programs means that you must understand the way in which processes and threads execute. This chapter introduces this topic and explains the way you tell the CLI which processes and threads it should apply a command to.

Chapter 4: Using the CLI

The CLI commands execute within the Tcl and TotalView environments. (The code used by the CLI and TotalView that interacts with your programs is shared.) This chapter explains the general Tcl environment and how you debug programs using the CLI.

Chapter 5: Type Transformations

There are times when TotalView cannot correctly format your data. For example, TotalView can have difficulty with the C++ STL. You can solve this problem by creating Tcl callbacks that tell the TotalView GUI how it should display information.

Chapter 6: CLI Commands

This chapter contains the **man** pages for CLI commands.

Appendix A: CLI Command Summary

This appendix contains a listing of all CLI commands, a brief explanation of what the command does, and a depiction of the command's syntax.

Appendix B: CLI Command Default Arena Widths

Here you will find a table containing the default focus for each command. (The *focus* indicates the processes and threads upon which a command acts.)

Appendix C: Distributed Array Type Mapping

This appendix contains a listing of an example program that creates a Mandlebrot set and type mapping for this program.

Conventions

The following table describes the conventions used in this book:

TABLE 1: **Book Conventions**

Convention	Meaning
|	Choose one of the listed commands. (| means "or".)
[ ]	Brackets are used when describing parts of a command that are optional. Be careful to distinguish between brackets used in command descriptions and brackets used by Tcl that evaluate expressions.
{ }	Braces are used when describing parts of a command where you must choose one of the options.
arguments	Within a command description, text in italic represent information you type. Elsewhere, italic is used for emphasis. You will not have any problems distinguishing between the uses.
Dark text	Within a command description, **dark text** represent key words or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.

Reporting Problems

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Our Internet E-Mail address is **support@etnus.com**
Call: 1-800-856-3766 in the United States
(+1) 508-652-7700 worldwide

If you are reporting a problem, please include the following information:

- The **version** of TotalView and the **platform** on which you are running TotalView

- An **example** that illustrates the problem
- A **record** of the sequence of events that led to the problem

See the TOTALVIEW RELEASE NOTES for complete instructions on how to report problems.

Chapter 2

A Few CLI/Tcl Macros

You can use the CLI in two ways—and, of course, you can combine these two ways. The first is as a command-line debugger that works with the TotalView Graphical User Interface (GUI) debugger. The second is as a debugging programming language that allows you to add your own commands and functions.

This chapter contains a few macros that show how the CLI programmatically interacts with your program and with TotalView. Reading a few examples without bothering too much with details will give you an appreciation for what the CLI can do and how it is used. As you will see, you really need to have a basic knowledge of Tcl before you can make full use of all CLI features.

The chapter presents the following macros:

- Setting the EXECUTABLE_PATH State Variable
- Initializing an Array Slice
- Printing an Array Slice
- Writing an Array Variable to a File
- Setting Breakpoints

Setting the EXECUTABLE_PATH State Variable

The following macro recursively descends through all directories starting at a location that you enter. (This is indicated by the *root* argument.) The macro will ignore directories named in the *filter* argument. The result is then set as the value of the CLI **EXECUTABLE_PATH** state variable.

```
# Usage:
#
#   rpath <root> <filter>
#
# If <root> is not specified, start at the current directory.
#
# <filter> is a regular expression that removes unwanted
# entries. If it is not specified, the macro automatically filters
# out CVS/RCS/SCCS directories.
#
# The TotalView search path is set to the result.

proc rpath { { root "." } { filter "/(CVS|RCS|SCCS)(/|$)" } } {

    # Invoke the UNIX find command to recursively obtain all
    # directory names below “root”.
    set find [split [exec find $root -type d -print] \n]

    set npath ""

   # Filter out unwanted directories.
    foreach path $find {
        if {! [regexp $filter $path] } {
            append npath “:”
            append npath $path
        }
    }

    # Tell TotalView to use it.
    dset EXECUTABLE_PATH $npath
}
```

In this macro, the final statement setting the **EXECUTABLE_PATH** state variable is the only statement that is unique to the CLI. All other statements are standard Tcl.

The **dset** command, like most CLI commands, begins with the letter **d**. (The **dset** command is only used in assigning values to CLI state variables. In contrast, values are assigned to Tcl variables by using the standard Tcl **set** command.)

Initializing an Array Slice

The following macro initializes an array slice to a constant value:

```
array_set (lower_bound upper bound var val) {
    for { set i $lower_bound } { $i <= $upperbound } { incr i } {
        dassign $var\($i) $val
    }
}
```

The CLI **dassign** command assigns a value to a variable. In this case, it is setting the value of an array element. Here is how you use this function:

```
d1.<> dprint list3
 list3 = {
    (1) = 1 (0x0000001)
    (2) = 2 (0x0000001)
    (3) = 3 (0x0000001)
 }
d1.<> array_set 2 3 list 3 99
d1.<> dprint list3
 list3 = {
    (1) = 1 (0x0000001)
    (2) = 99 (0x0000063)
    (3) = 99 (0x0000063)
 }
```

Printing an Array Slice

The following macro prints a Fortran array slice. This macro, like other ones shown in this chapter, relies heavily on Tcl and uses unique CLI commands sparingly.

```
proc pf2Dslice {anArray i1 i2 j1 j2 {i3 1} {j3 1} {width 20}} {
    for {set i $i1} {$i <= $i2} {incr i $i3} {
        set row_out ""
        for {set j $j1} {$j <= $j2} {incr j $j3} {
            set ij [capture dprint $anArray\($i,$j\)]
            set ij [string range $ij \
                [expr [string first "=" $ij] + 1] end]
            set ij [string trimright $ij]
```

```
            if { [string first "-" $ij] == 1} {
                set ij [string range $ij 1 end] }
            append ij "                    "
            append row_out " " [string range $ij 0 $width] " "
        }
        puts $row_out
    }
}
```

After invoking this macro, the CLI prints a two-dimensional slice (**i1:i2:i3, j1:j2:j3**) of a Fortran array to a numeric field whose width is specified by the **width** argument. This width does not include a leading minus (-) sign.

All but one line is standard Tcl. This line uses the **dprint** command to obtain the value of one array element. This element's value is then captured into a variable. (**dprint** does not return a value. The CLI **capture** command allows a value that is normally printed to be sent to a variable.)

Here are several examples:

```
d1.<> pf2Dslice a 1 4 1 4
          0.841470956802 0.909297406673 0.141120001673-0.756802499294
          0.909297406673-0.756802499294-0.279415488243 0.989358246326
          0.141120001673-0.279415488243 0.412118494510-0.536572933197
         -0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslice a 1 4 1 4 1 1 17
          0.841470956802 0.909297406673 0.141120001673-0.756802499294
          0.909297406673-0.756802499294-0.279415488243 0.989358246326
          0.141120001673-0.279415488243 0.412118494510-0.536572933197
         -0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslice a 1 4 1 4 2 2 10
          0.84147095 0.14112000
          0.14112000 0.41211849
d1.<> pf2Dslice a 2 4 2 4 2 2 10
         -0.75680249 0.98935824
          0.98935824-0.28790330
d1.<>
```

Writing an Array Variable to a File

There are many times when you would like to save the value of an array so that you can analyze its results at a later time. The following macro writes an array's value to a file and saves it.

```
proc save_to_file { var fname } {
    set values [capture dprint $var]
    set f [open $fname w]

    puts $f $values
    close $f
}
```

The following shows how you might use this macro. Notice that using the **exec** command lets **cat** display the file that was just written.

```
d1.<> dprint list3
 list3 = {
    (1) = 1 (0x00000001)
    (2) = 2 (0x00000002)
    (3) = 3 (0x00000003)
 }
d1.<> save_to_file list3 foo
d1.<> exec cat foo
 list3 = {
    (1) = 1 (0x00000001)
    (2) = 2 (0x00000002)
    (3) = 3 (0x00000003)
 }
d1.<>
```

Setting Breakpoints

In many cases, your knowledge of what a program is doing lets you make predictions as to where problems will occur. The following CLI macro parses comments that you can include within a source file and, depending on the comment's text, sets a breakpoint or an evaluation point.

Immediately following this listing is an excerpt from a program that uses this macro.

```
# make_actions: Parse a source file, and insert
# evaluation and breakpoints according to comments.
#
proc make_actions { { filename "" } } {
```

```
if { $filename == "" } {
    puts "You need to specify a filename"
    error "No filename"
}

        # Open the program's source file and initialize a
        # few variables.
set fname [set filename]
set fsource [open $fname r]
set lineno 0
set incomment 0

        # Look for "signals" that indicate the kind of action
        # point; they are buried in the C comments.
while { [gets $fsource line] != -1} {
    incr lineno
    set bpline $lineno

            # Look for a one-line evaluation point. The
            # format is ... /* EVAL: some_text */.
            # The text after EVAL and before the "*/" in
            # the comment is assigned to "code".
    if [regexp "/\\* EVAL: *(.*)\\*/" $line all code] {
        dbreak $fname\#$bpline -e $code
        continue
    }

            # Look for a multiline evaluation point.
    if [regexp "/\\* EVAL: *(.*)" $line all code] {
            # Append lines to "code".
        while { [gets $fsource interiorline] != -1} {
            incr lineno

            # Tabs will confuse dbreak.
            regsub -all \t $interiorline "   " interiorline

            # If "*/" is found, add the text to "code", then
            # leave the loop. Otherwise, add the text, and
            # continue looping.
            if [regexp "(.*)\\*/" $interiorline all interiorcode]{
                append code \n $interiorcode
                break
```

```
                } else {
                    append code \n $interiorline
                }
            }
            dbreak $fname\#$bpline -e $code
            continue
        }
            # Look for a breakpoint.
        if [regexp "/\* STOP: .*" $line] {
            dbreak $fname\#$bpline
            continue
        }
            # Look for a command to be executed by Tcl.
        if [regexp "/\* *CMD: *(.*)\*/" $line all cmd] {
            puts "CMD: [set cmd]"
            eval $cmd
        }
    }
    close $fsource
}
```

The only similarity between this example and the previous two is that almost all of the statements are Tcl. The only purely CLI commands are the instances of the **dbreak** command. (This command sets evaluation points and breakpoints.)

The following excerpt from a larger program shows how you would embed comments within a source file that would be read by this macro:

```
...
struct struct_bit_fields_only {
    unsigned f3 : 3;
    unsigned f4 : 4;
    unsigned f5 : 5;
    unsigned f20 : 20;
    unsigned f32 : 32;
} sbfo, *sbfop = &sbfo;
...
int main()
{
    struct struct_bit_fields_only *lbfop = &sbfo;
...
```

```
    int i;
    int j;
    sbfo.f3 = 3;
    sbfo.f4 = 4;
    sbfo.f5 = 5;
    sbfo.f20 = 20;
    sbfo.f32 = 32;
...
    /* TEST: Check to see if we can access all the values */
    i=i;    /* STOP: // Should stop */
    i=1;    /* EVAL: if (sbfo.f3 != 3) $stop; // Should not stop */
    i=2;    /* EVAL: if (sbfo.f4 != 4) $stop; // Should not stop */
    i=3;    /* EVAL: if (sbfo.f5 != 5) $stop; // Should not stop */
    ...
    return 0;
}
```

The **make_actions** macro reads a source file one line at a time. As it reads these lines, the regular expressions look for comments that begin with **/* STOP**, **/* EVAL**, and **/* CMD**. After parsing the comment, it sets a breakpoint at a *stop* line, an evaluation points at an *eval* line, or executes a command at a *cmd* line.

Using evaluation points can be confusing because evaluation point syntax differs from that of Tcl. In this example, the **$stop** command is a command contained within TotalView (and the CLI). It is not a Tcl variable. In other cases, the evaluation statements will be in the C or Fortran programming languages.

Chapter 3

Groups, Processes, and Threads

While the specifics of how multiprocess, multithreaded programs execute differ greatly, all share some general characteristics. This chapter defines how TotalView looks at processes and threads. It also describes the way in which you tell the CLI which processes and threads it should direct a command to.

A Couple of Processes

When programmers write single-threaded, single-process programs, they can almost always answer the question "Do you know where your program is?" These kind of programs are rather simple, looking something like this:

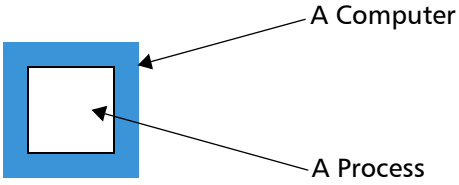

FIGURE 1: **A Uniprocessor**

If you use a debugger, any debugger, on something like this, you can almost always figure out what is going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, and then inspect variables to see what they have been set to. If you suspect there is a logic problem, you can step the program through its statements, seeing what happens and where things are going wrong.

What is actually occurring is a lot more complicated than this as your computer is executing a great number of programs. For example, your computing environment could have daemons and other support programs executing.

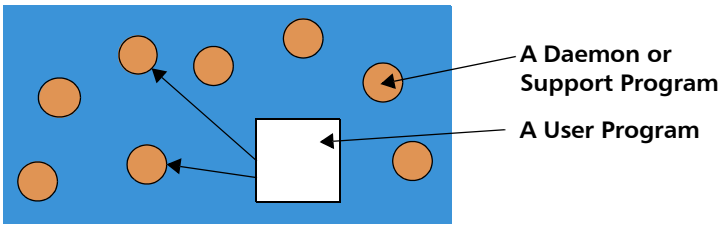

FIGURE 2: **A Program and Daemons**

These additional processes simplify a programmer's life because the application program no longer had to do everything itself. It could hand some things off to another program or process and it would do the program's bidding.

Figure 2 assumes that the application program only sends requests to the daemon. More complicated architectures are quite common. For example, the following figure shows an E-mail program communicating with a daemon on its computer. After receiving a request, this daemon sends data to an E-mail daemon on another computer which then delivers the data.

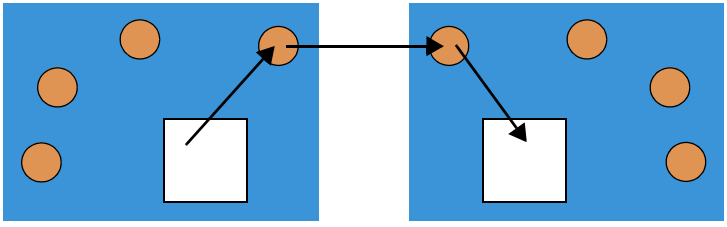

FIGURE 3: **Mail with Daemons**

This kind of processing assumes that the jobs are disconnected. That is, no real cooperation exists between the processes. In all cases, one program hands work to another. After the handoff occurs, there is no more interaction. While this is an extremely useful model, a more general model is one

where a program divides its work, parceling it out to other computers. When this occurs, one program relies on another program to do some of its work. To gain any advantage, however, the work being sent to the second computer has to be work that the first computer does not need right away. In this way, the two computers act more or less independently. And, because the first computer did not have to do the work that the second computer did, the program could complete faster. (See Figure 4.)

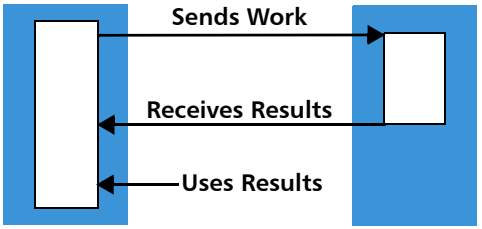

FIGURE 4: **Two Computers Working on One Problem**

Here is one of the problems with this scenario: because programs have bugs, how does a programmer debug what is happening on the second computer? A horrid solution is to have a debugger running on each computer. A slightly better solution is to create a program the same way as was done with one computer, get it working, and then split it up so it could use more than one computer. If done this way, there is a likelihood that any problems that occur will occur in the code that splits up the problem.

The TotalView solution is even better. It places a server on each processor as a program is launched. The server then communicates with the "main" TotalView. This debugging architecture gives you one central location from which you can manage and examine all aspects of your program.

Some Threads

The support programs just discussed are owned by the operating system. These programs execute a variety of activities from managing computer resources to providing services such as printing. If the operating system can have many independently operating components, why can't a program?

One programming model splits the work off into somewhat independent tasks within the same process. This is the *threads* model. (See Figure 5.) This figure also shows, for the last time, the daemon processes that are executing. From now on, just assume that they are there.

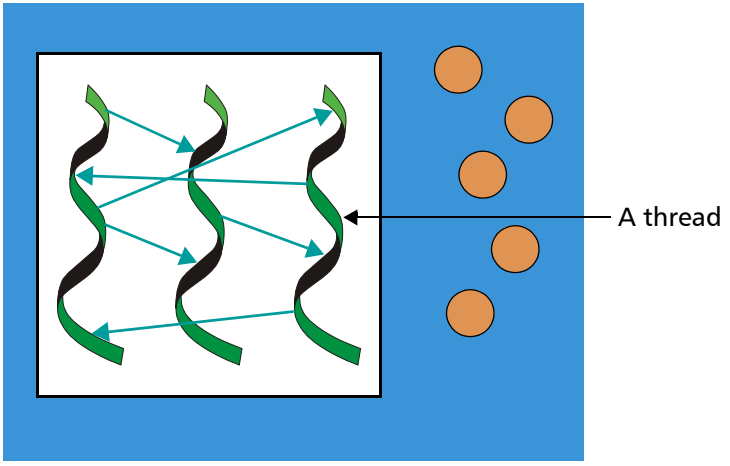

FIGURE 5: **Threads**

In this computing model, a program (the main thread) creates threads and these threads can also create threads if they need to. Each thread executes relatively independently from other threads.

The debugging problem here is similar to the problem of processes running on different machines. In both cases, a debugger has to intervene with anything that is executing.

In the examples used so far, each executing process and thread is doing something different and, except for when you need one thread to wait for another, it is hard to know what any thread is doing. And, the nature of these kinds of programs prevents you from running your program single-threaded; that is, having only one thing running at a time.

Even More Complicated Programming Models

In the same way that software computing architectures become more complicated, advances in hardware design placed more than one processor within a computer. So, expanding on what is shown in Figure 4, you could be writing programs that execute in environments like what is shown in Figure 6.

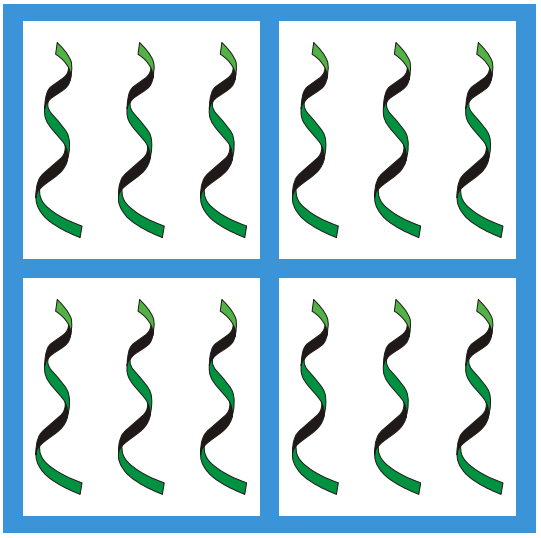

FIGURE 6: **Four-Processor Computer**

This figure shows four linked processors in one computer, each of which has three threads. This architecture could, in one sense, be thought of as an extension to the model having more than one computer. And, if you think of your architecture like this, there is no reason that you can not join many computers together to solve problems. Figure 7 shows five computers, each with four processors. Every program running on every processor has three threads, which means that altogether there are 60 user threads.

This figure depicts only processors and threads. It does not have any information about the nature of the programs and threads or even if the programs are the same or are different.

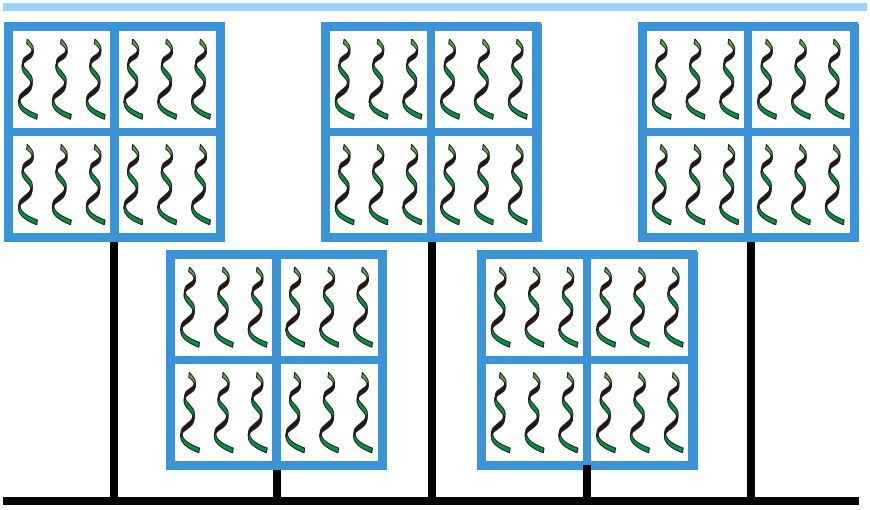

FIGURE 7: **Four Processor Computer Networks**

At any time, it is next to impossible to guess which threads are executing and what a thread is actually executing. To make matters worse, many multiprocessor programs begin by invoking a process such as **mpirun** or IBM's POE whose function is to distribute and control the work being performed. In this kind of environment, a program (or the program within a library) is using another program to control how it distributes work across processors.

If everything goes right, life is good. When there are problems—and there are always problems—traditional debuggers and solutions are helpless. As you will see, TotalView organizes this mass of executing procedures for you and, because operating systems can complicate things greatly, TotalView lets you distinguish between threads and processes used by the operating system and those used by your program.

More on Threads

All threads aren't the same. Figure 8 shows a program with three threads.

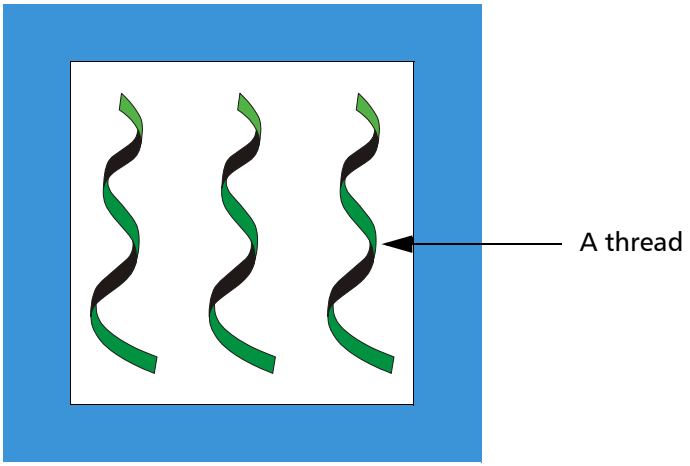

FIGURE 8: **Threads**

For the moment, assume that all of these threads are "user threads"; that is, they are threads that perform some activity that you have programmed.

NOTE Many computer architectures have something called "user mode," "user space", or something similar. "User threads" means something else. Without trying to be rigorous, the TotalView definition of a "user thread" is simply a unit of execution created by a program.

Other threads can also be executing. For example, the threads that are part of the operating environment are "*manager threads*". Things would be nice and easy if this was all there was to it. Unfortunately, all threads are not created equal and all threads do not execute equally. In most cases, a program creates manager-like threads. In Figure 8, the horizontal threads at the bottom of the figure are user-created manager threads.

As these user-created manager threads are designed to perform a service for other threads in the program, they can also be called "*service threads*".

One reason you need to know which of your threads are service threads is that this kind of thread performs different kinds of activities from other user threads. Because their activities are so different, they are usually developed separately and are not involved with the fundamental problems being solved by the program. For example, a service thread that dispatches

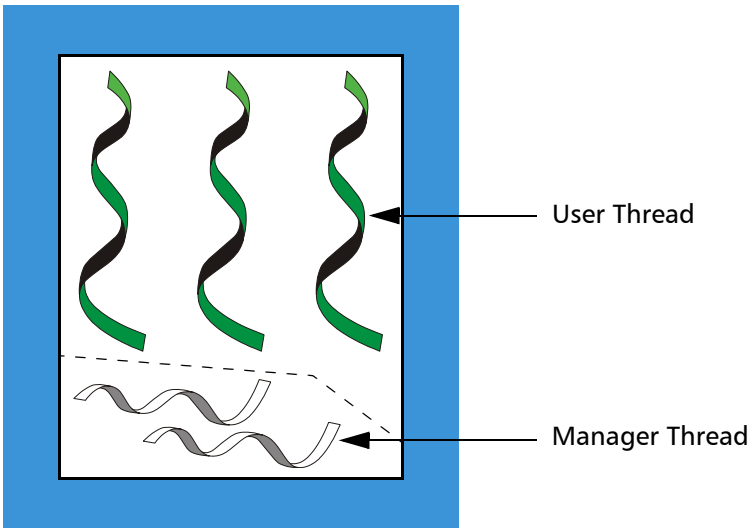

FIGURE 9: **User Threads and Service Threads**

messages sent from other threads may have bugs, but the bugs are of a different kind and the problems they have can most often be dealt with separately from bugs that would occur in non service user threads.

A second reason is that you do not want to wait for these threads when stepping your program because these threads will never get there.

In contrast, your user threads are the agents performing the actual work, and the interactions among them are where the action is. Being able to distinguish between the two kinds of threads means that you can focus on the threads and processes that are actively participating in an activity, rather than those sitting back and performing more subordinate activities.

Types of User Threads

Sometimes, TotalView can identify which threads are performing service activities. In other cases, this is not possible. While some threads can be identified (and on some architectures this may not be possible), you are often forced to identify which user threads are performing service activities.

In Figure 10, one of the three nonmanager worker threads performs a different kind of activity than the other two threads. As an example, this

could be a thread whose sole function is to send data to a printer in response to a request from either of the other two threads.

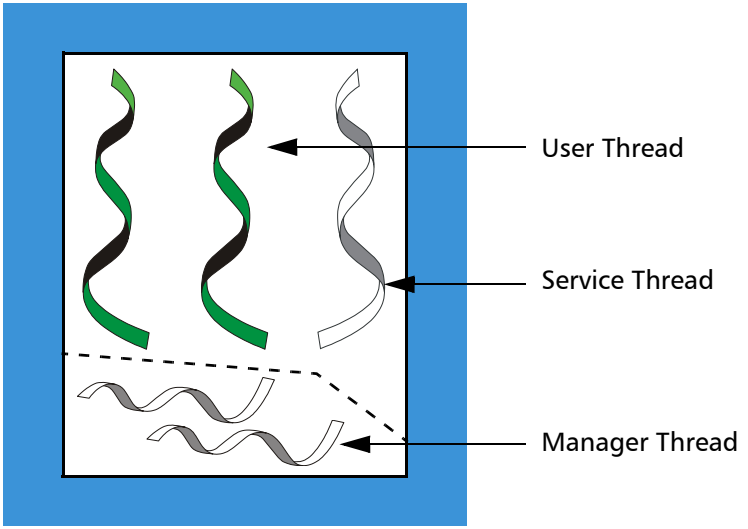

FIGURE 10: **User, Service, and Manager Threads**

So, while this figure shows five threads, most of your efforts will revolve around just two threads. These threads two threads are called *worker* threads.

Organizing Chaos

While it is possible to attack the kinds of programs that are running thousands of processes across hundreds of computers one-at-a-time, it is not very practical. What TotalView does is structure these processes for you and then let you reorganize this information into "groups". Here are quick definitions of the four kinds of groups:

- **Control Group**: All the processes created by a program running on all processors. That is, the control group includes all processes that are running on multiple processors. If your program uses processes that it did not create, these other processes are in another control group.

- **Share Group**: All the processes within a control group that share the same code. In most cases, your program will have more than one share group. Share groups, like control groups, can have processes that execute on more than one processor.
- **Workers Group**: All the worker threads within a control group. These threads can be drawn from more than one share group.
- **Lockstep Group**: All threads that are at the same PC (program counter). This group is a subset of a workers group. Because all threads execute asynchronously, a lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group.

In the list, the first two groups contain processes and the last two groups contain threads. And, notice that "same code" means that the processes have the same executable file name.

TotalView's commands let you manipulate processes and threads individually and by groups. In addition, you can create your own groups and manipulate a group's contents (to some extent).

NOTE Not all operating systems let you individually run a thread.

Figure 11 shows a processor running five processes (ignoring daemons and other programs not related to your program) and the threads within them. The figure indicates a control group and two share groups.

The elements in this figure are as follows:

CPU	Everything represented by this drawing exists within one processor.
Processes	Processes being executed by the CPU.
Control Group	The five processes make up the control group. This diagram does not indicate which process is the **main** procedure.
Share Groups	The control group has two share groups. The three processes in the first share group have the same executable. The two processes in the second share group share a second executable.

Figure 12 looks at how the threads in this drawing are organized. As you can see, this figure adds the workers group and two lockstep groups.

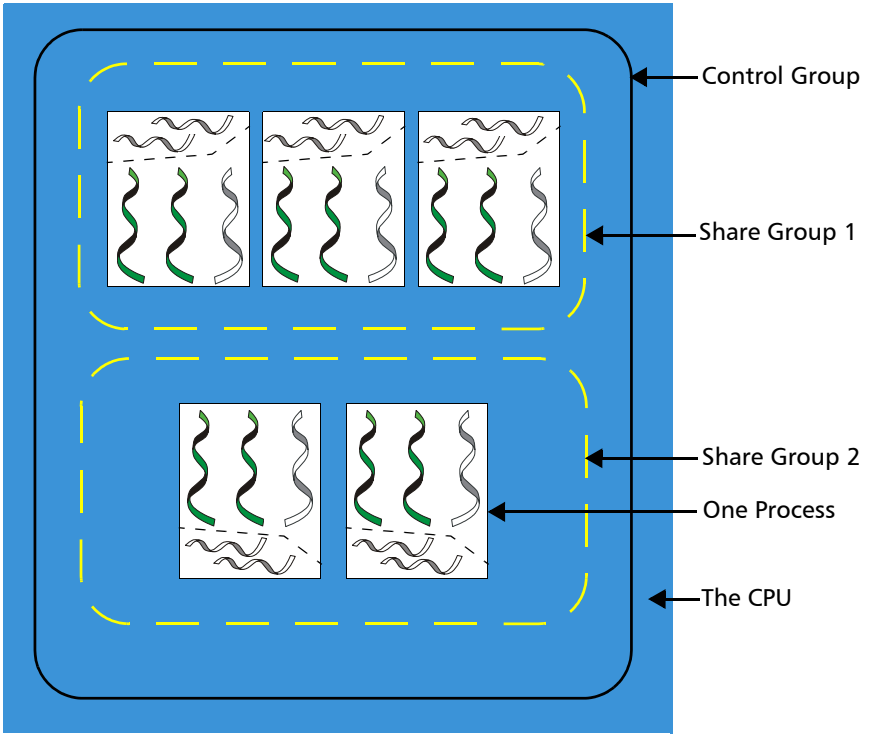

FIGURE 11: **Five Processors and Processor Groups**

NOTE The control group is not shown as it encompasses everything in Figure 12.

Here is a description of the added elements in this figure:

Workers Group All non-manager threads within the control group make up the workers group. Notice that this group includes service threads.

Lockstep Group Each share group has its own lockstep groups. Graphically, two lockstep groups are indicated, one in each share group.

If other threads are stopped, this picture indicates that they are not participating in either of these two lockstep groups. Recall that a stopped thread is always in a lockstep group. (It's OK if a lockstep group has only one member.)

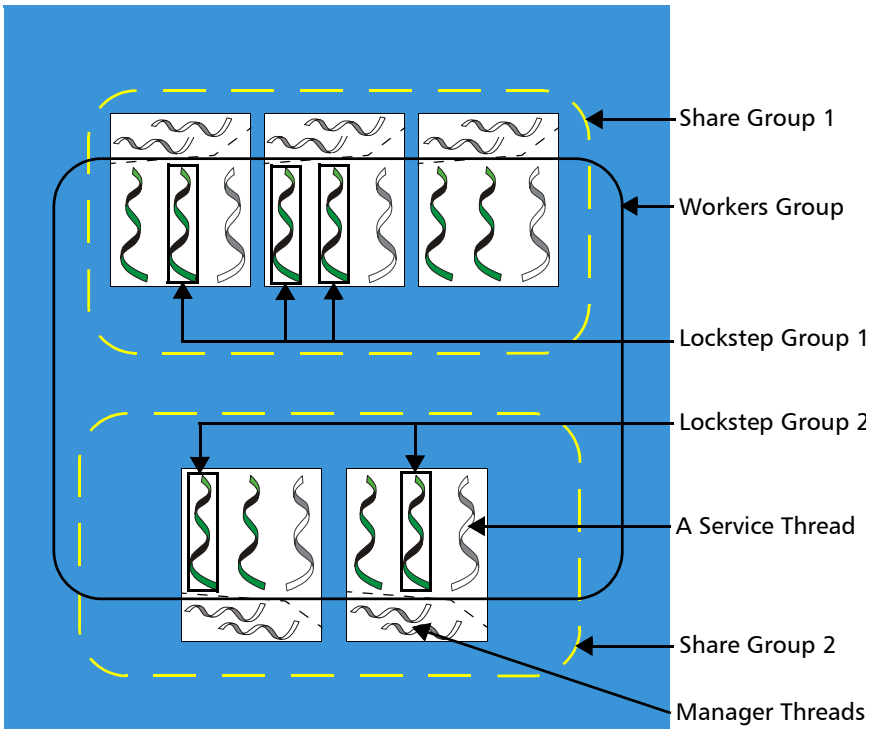

FIGURE 12: **Five Processors and Processor Groups**

Service Threads Each process has one service thread. A process can have any number of service threads. This figure, however, only shows one.

Manager Threads The only threads that are not participating in the workers group are the ten manager threads.

Figure 13 extends the previous figure to show the same kinds of information executing on two processors.

This figure differs from the one it is based on in that it has ten processes executing within two processors rather than five processes within one processor. Although the number of processors has changed, the number of control and share groups is unchanged. This is not to say that the number of groups could not be different. It's just they are not in this example.

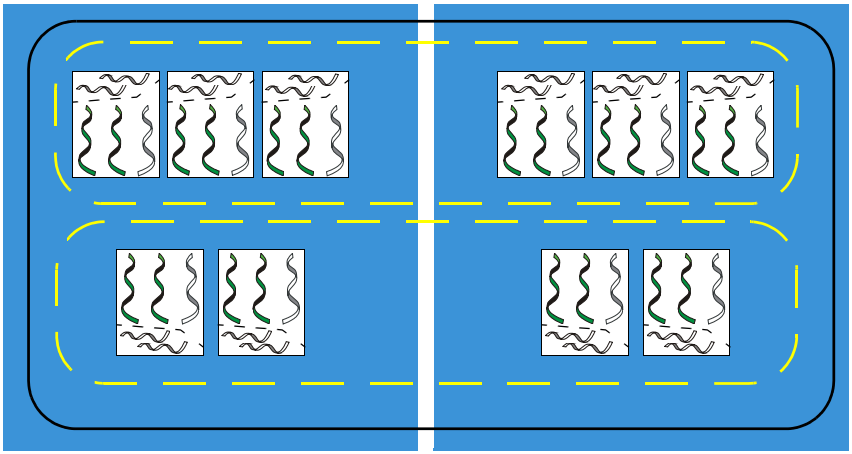

FIGURE 13: **Five Processes and Their Groups on Two Computers**

Creating Groups

TotalView automatically creates and places items in groups as they are created. The exception is the lockstep groups that are created or changed whenever a program hits an action point or is stopped for any reason. While there are many ways that this kind of organization can be built up, the following steps indicate the beginnings of how this might occur:

1 TotalView and your program are launched and your program begins executing within TotalView.

Control group: A group is created as the program is loaded.

Share group: A group is created as the program begins executing.

Workers group: The thread in the main routine is the workers group.

Lockstep group: There is no lockstep group because the thread is running.

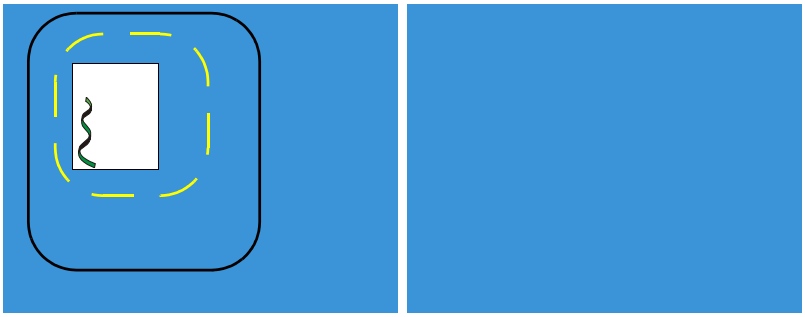

FIGURE 14: **Step 1: A Program Starts**

2 The program forks a process.

Control group: A second process is added to the existing group.

Share group: A second process is added to the existing group.

Workers group: TotalView adds the thread in the second process to the existing group.

Lockstep group: There are no lockstep groups because the threads are running.

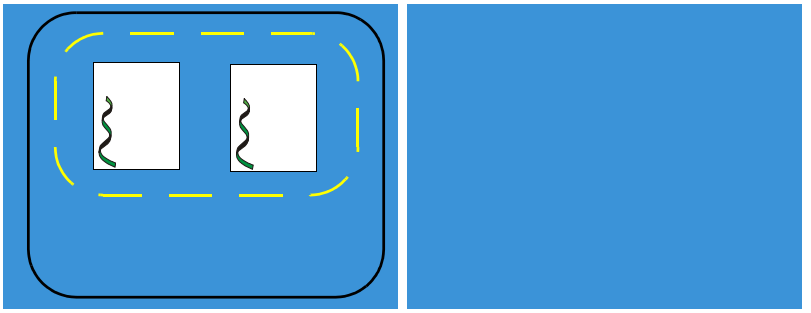

FIGURE 15: **Step 2: Forking a Process**

3 The second process is exec'd.

Control group: The group is unchanged.

Share group: A second share group is created having this exec'd process as a member. This process is removed from the first share group.

Workers group: Both threads are in the worker s group.

Lockstep group: There are no lockstep groups because the threads are running.

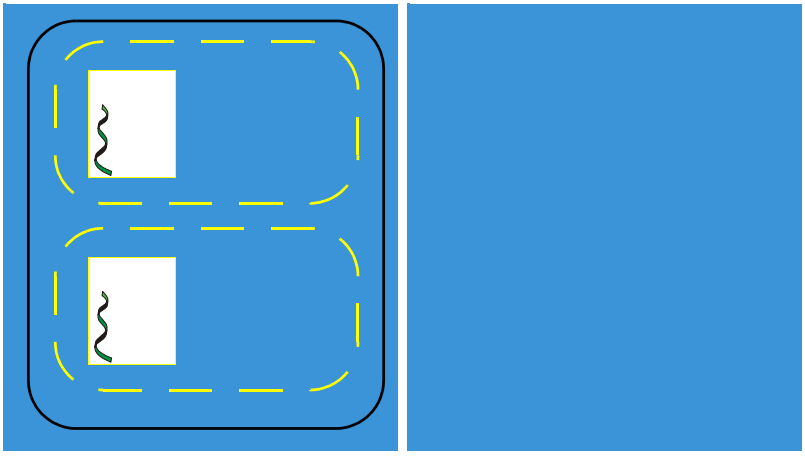

FIGURE 16: **Step 3: Exec'ing a Process**

4 The first process hits a break point.

Control group: The group is unchanged.

Share group: The groups are unchanged.

Workers group: The group is unchanged.

Lockstep group: A lockstep group is created whose member is the thread of the current process. (In this example, each thread is its own lockstep group.)

5 The program is continued and a second version of your program is started from the shell. You attach to it within TotalView and put it in the same control group as your first process.

Control group: A third process is added.

Share group: This third process is added to the first share group.

Workers group: The thread in this third process is added to the group.

Lockstep group: There are no lockstep groups because the threads are running.

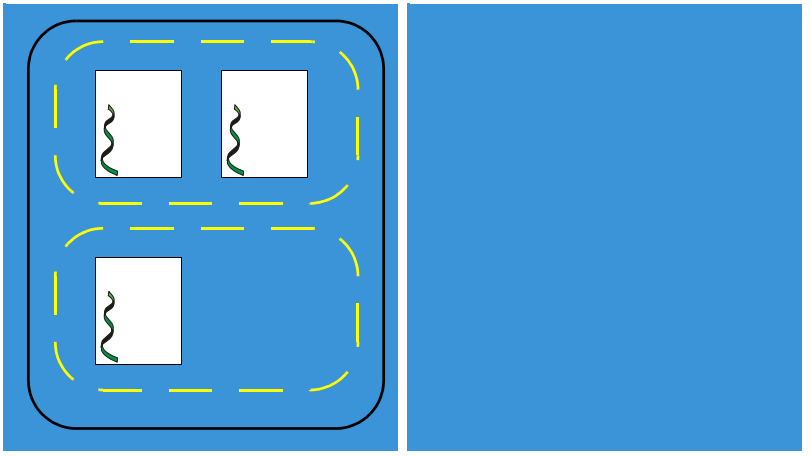

FIGURE 17: **Step 5: Creating a Second Version**

6 Your program creates a process on another computer.

Control group: The control group is extended so that it contains this fourth process that is running on the second computer.

Share group: The first share group now contains this newly created process even though it is running on the second computer.

Workers group: The thread within this fourth process is added to the workers group.

Lockstep group: There are no lockstep groups because the threads are running.

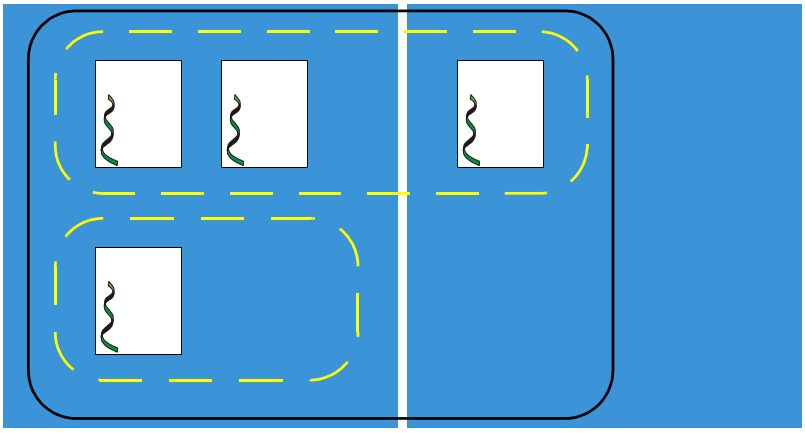

FIGURE 18: **Step 6: Creating a Remote Process**

7 A process within control group 1 creates a thread. This adds a second thread to one of the processes.

Control group: The group is unchanged.

Share group: The group is unchanged.

Workers group: A fourth thread is added to this group.

Lockstep group: There are no lockstep groups because the threads are running.

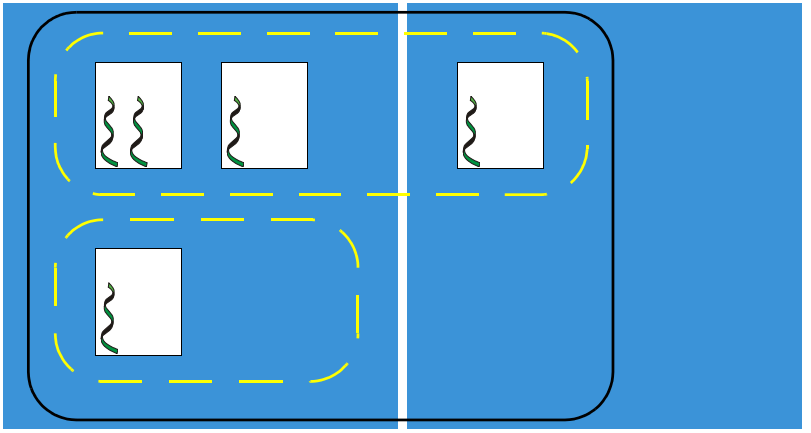

FIGURE 19: **Step 7: A Thread Is Created**

8 A breakpoint is set on a line within a process executing in the first share group and the breakpoint is shared. The process executes until all three processes are at the breakpoint.

Control group: The group is unchanged.

Share group: The groups are unchanged.

Workers group: The group is unchanged.

Lockstep groups: Lockstep groups are created whose members are the four threads in the first share group.

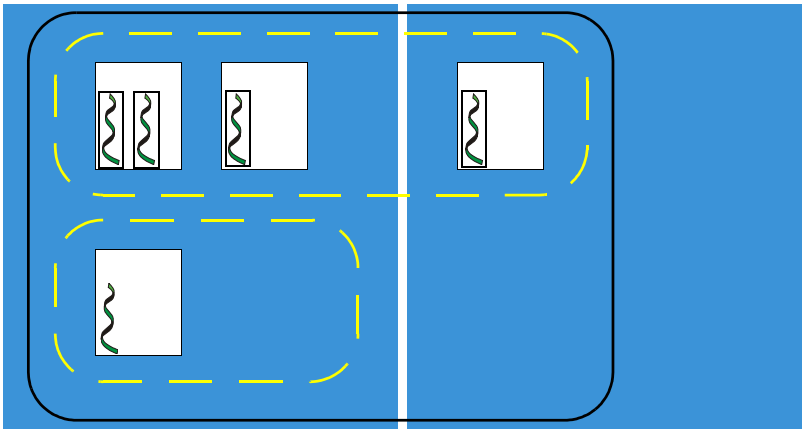

FIGURE 20: **Step 8: Hitting a Breakpoint**

9 You tell TotalView to step the lockstep group.

Control group: The group is unchanged.

Share group: The groups are unchanged.

Workers group: The group is unchanged.

Lockstep group: The lockstep groups are unchanged. (Note that there are other lockstep groups.)

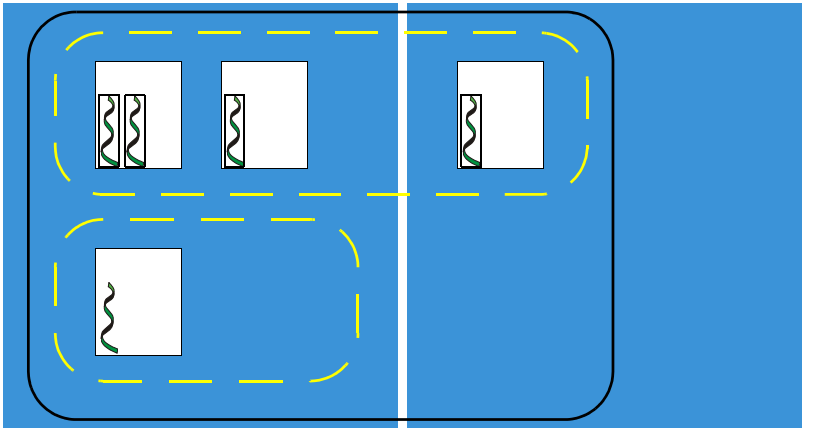

FIGURE 21: **Step 9: Stepping the Lockstep Group**

Clearly, this example could keep on going until a much more complicated system of processes and threads was created. However, it should give you an idea of what is occurring.

Simplifying What You're Debugging

It's time to say something pretty simple: the reason you are using a debugger is because your program isn't operating correctly and the way you think you're going to solve the problem (unless it is a &%$# operating system problem, which, of course, it usually is) is by stopping your program's threads, examining the values assigned to variables, and stepping your program so you can see what is happening as it executes.

Unfortunately, your multiprocess, multithreaded program and the computers upon which it is executing have lots of things executing that you want

TotalView to ignore. For example, you don't want to be examining manager and service threads created by the operating system, your programming environment, and your program.

Also, most of us are incapable of understanding exactly how a program is acting when perhaps thousands of processes are executing asynchronously. Fortunately, there are only a few problems that require full asynchronous behavior.

One of the first simplifications you can make is to change the number of processes. For example, suppose you have a buggy MPI program running on 100 processors. Your first step might be to have it execute in a 4-processor environment.

After you get the program running under TotalView's control, you will want to run the process being debugged to an action point, so you can inspect the program's state at that place. In many cases, because your program has places where processes are forced to wait for an interaction with other processes, you can ignore what they are doing.

NOTE TotalView lets you control as many groups, processes, or threads as you need to control. While each can be controlled individually, you will probably have problems remembering what you're doing if you're controlling large numbers of these things. The reason that TotalView creates and manages groups is so that you can focus on portions of your program.

In most cases, you do not need to interact with everything that is executing. Instead, you want to focus on one process and the data that this process is manipulating. Things get complicated when the process being investigated is using data created by other processes, and these processes may have dependencies on other processes.

All this means that there is a rather typical pattern to the way you use TotalView to locate problems:

1 At some point, you should make sure that the groups you are manipulating do not contain service or manager threads. (You can remove processes and threads from a group with the **dgroups –remove** command.)

2. Place an action point within a process or thread and begin investigating the problem. In many cases, you are setting an action point at a place where you hope the program is still executing correctly. Because you are debugging a multiprocess, multithreaded program, you want to set a barrier point so that all threads and process are at the same place.
3. After execution stops at the barrier point, look at the contents of your variables. Verify that your program state is actually correct.
4. Begin stepping your program through its code. In most cases, step your program synchronously stepping or set barriers so that everything isn't running freely.
5. Here's where things begin to get complicated. You've been focusing on one process or thread. If another process or thread is modifying the data and you become convinced that this is the problem, you'll want to go off to it and see what is going on.

The trick here, and it really isn't much of a trick, is keeping your focus narrow, so that you're just investigating a limited number of behaviors. This is where debugging becomes an art. A multiprocess, multithreaded program can be doing a great number of things. Understanding where to look when problems occur is the "art".

For example, you'll most often want to execute commands at the default focus. Only when you think that the problem is occurring in another process will you change to that process. You'll still be executing in a default focus, but this time the default focus is focussed at this other process.

In contrast, while you will often want to do something using another focus, what you will probably do is:

- Modify the focus so that it affects just the next command. For example, here's the command that steps thread 7 in process 3:

  dfocus t3.7 dstep

  (In this example, the **dfocus** directive tells TotalView to limit the scope of what it does for the command that immediately follows and then, after the command completes, to restore the old focus.)
- Use the **dfocus** command to change focus temporarily, execute a few commands, and then return to the original focus.

Setting Process and Thread Focus

When the CLI executes a command, TotalView must decide which processes and threads it should act upon. Most commands have a default set of threads and processes and, in most cases, you won't want to change the default. There are times, however, when you'll need to change this default. This section begins a rather intensive look at how you tell TotalView what processes and threads it should use as the target of a command.

Process/Thread Sets

All CLI commands operate upon a set a processes and threads. This set is called a P/T (*Process*/T*hread*) *set*. A P/T set is a Tcl list containing one or more P/T identifiers. (The next section explains what a P/T identifier is.) Tcl lets you create lists in two ways:

- You can enter these identifiers within braces (**{ }**).
- You can use Tcl commands that create and manipulate lists.

These lists are then used as arguments to a CLI command. If you are entering one element, you usually do not have to use Tcl's list syntax.

For example, the following list contains specifiers for process 2, thread 1 and process 3, thread 2:

```
{ p2.1 p3.2 }
```

Unlike a serial debugger where each command clearly applies to the only executing thread, the CLI can control and monitor many threads and many different locations. The P/T set indicates the groups, processes, and threads that are the target of the CLI command. No limitation exists on the number of groups, processes, and threads within a set.

If you do not explicitly specify a P/T set, the CLI defines a target set for you. This set is displayed as the (default) CLI prompt. (For information on this prompt, see "Command and Prompt Formats" on page 81.)

You can change the focus upon which a command acts by using the **dfocus** command. If the CLI executes **dfocus** as a separate command, it changes

the default P/T set. For example, if the default focus is process 1, the following command changes the default focus to process 2:

```
dfocus p2
```

After the CLI executes this command, the commands that it will now execute will focus on process 2.

If you type the **dfocus** command as part of another command, the CLI changes the target for just the command that follows. After the command executes, the *old* default is restored.

The following example shows both of these ways to use the **dfocus** command. Assume that the current focus is process 1, thread 1. The following commands change the default focus to group 2 and then step the threads in this group twice:

```
dfocus g2
dstep
dstep
```

Before the **dstep** command executes, it looks for the thread of interest—the thread that was the focus of activity—in group 2. TotalView will then step all threads in the same lockstep group as the thread of interest.

In contrast, the following commands step group 2 and then step process 1, thread 1:

```
dfocus g2 dstep
dstep
```

This is because the **dfocus** command is used as a modifier instead of as a stand-alone command.

Some commands can only operate at the process level—that is, you cannot apply them to a single thread (or group of threads) in the process but must apply them to all or to none.

Arenas

A P/T identifier often indicates a number of groups, processes, and threads. For example, assume that two threads executing the same executable image in process 2 are stopped at the same statement. This means

that TotalView places the two stopped threads into a lockstep group. If the default focus is process 2, stepping this process actually steps both of these threads.

The CLI uses the term *arena* to define the processes and threads that are the target of an action. In this case, the arena has two threads. Many CLI commands can act upon one or more arenas. For example, here is a command with two arenas:

```
dfocus { p1 p2 }
```

The two arenas are process 1 and process 2.

GOI, POI, and TOI

You will start see three terms being used:

- GOI, which means Group of Interest
- POI, which means Process of Interest
- TOI, which means Thread of Interest

When the CLI executes a command, the arena decides the scope of what will run. It does not, however, determine what will run. Depending upon the command, the CLI looks determines what is the TOI, POI, or GOI, then executes upon that thread, process, or group. For example, you tell the CLI to step the current control group. In this case, it needs to know what the TOI is so it can determine what the lockstep group. (You can only step a lockstep group.) The lockstep group, which is a thread group, is part of a share group, which is a process group. This share group is part of a control group. So, know it know what the GOI is. This is important because, as you will see, while it now knows what it will step (the threads in the lockstep group), it also knows what it will allow to run freely while it is stepping these threads.

Using the GOI, POI, and TOI will become clearer as you read the rest of this chapter.

Specifying Processes and Threads

A previous section said that a P/T set is a list. This ignored what the individual elements of the list are. A better definition is that a P/T set is a list of arenas, where an *arena* is the processes, threads, and groups that are affected by a CLI debugging command. Each *arena specifier* describes a single arena in which a command will act; the *list* is just a collection of arenas. Most commands iterate over the list, acting individually on an arena. Some output commands, however, may combine the arenas and act on them as a single target.

An arena specifier includes a *width* and a *thread of interest*. ("Widths" are discussed later in this section.) Within the P/T set, the *thread of interest* specifies a target thread, while the width specifies how many threads surrounding the thread of interest are affected.

The Thread of Interest

The thread of interest is specified as **p.t,** where **p** is the TotalView process ID (PID) and **t** is the TotalView thread ID (TID). The **p.t** combination identifies the process and thread of interest. The thread of interest is the primary thread affected by a command. This means that it is the primary focus for a CLI command. For example, the **dstep** command always steps the thread of interest, but it may optionally run the rest of the threads in the process of interest and may step other processes in the group.

The CLI has two symbols with special meaning when specifying P/T sets:

- The less-than (**<**) character in place of the TID to indicate the *lowest number worker thread* in the process. If, however, the arena explicitly names a thread group, **<** means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which may not be thread 1; for example, the first and only user thread may be thread number 3 on Compaq systems.
- A dot (.) indicates the current set. While this is seldom needed interactively, it can be useful in scripts.

Process and Thread Widths

You can enter a P/T set in two ways. If you are not manipulating groups, the format is:

[*width_letter*][PID][.TID]

NOTE The next section extends this format to include groups.

For example, **p2.3** indicates process 2, thread 3.

While the syntax seems to indicate that you do not need to enter any element, the CLI requires that, at a minimum, you enter one of the three components. Because the CLI has an extensive set of defaults, it will try to fill in what you omit. The only requirement is that when you use more than one element, you use it in the order shown in this representation.

The *width_letter* indicates which processes and threads are part of this arena specifier. These letters are:

t *Thread width*

A command's target is the indicated thread.

p *Process width*

A command's target is the process containing the thread of interest.

g *Group width*

A command's target is the group containing the process of interest.

a *All processes*

A command's target is all threads in the group of interest that are in the process of interest.

d *Default width*

A command's target depends on the default for each command. This is also the width to which the default focus is set. For example, the **dstep** command defaults to process width (run the process while stepping one thread), and the **dwhere** command defaults to thread width (backtrace just one thread). Default widths are listed in "CLI Command Default Arena Widths" on page 277.

These widths must be entered as lowercase letters.

The following figure illustrates the relationship of these specifiers:

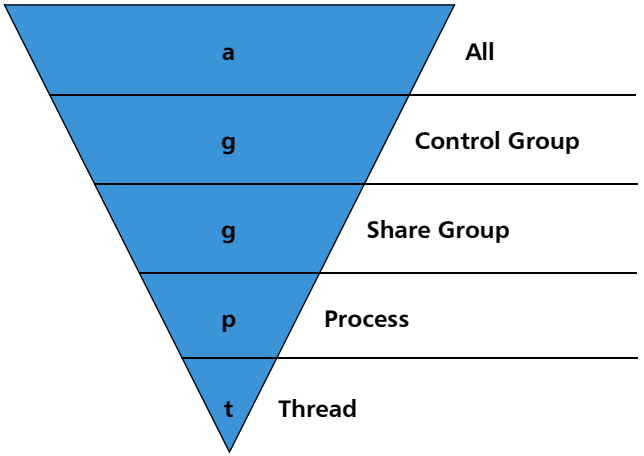

FIGURE 22: **Width Specifiers**

NOTE Notice that the "g" specifier indicates control and share groups.

You can visualize this relationship as a triangle with its base on the top. This indicates that the arena focuses on a greater number of entities as you move from thread level at the bottom to "all" level at the top.

As mentioned previously, the thread of interest specifies a particular target thread, while the width specifies how many threads surrounding the thread of interest are affected. For example, the **dstep** command always requires a thread of interest, but entering this command can:

- Step just the thread of interest during the step operation (single-thread single-step).
- Step the thread of interest and step all threads in the process containing the thread of interest (process-level single-step).
- Step all processes in the group that have threads at the same PC (program counter) as the thread of interest (group-level single-step).

This list doesn't indicate what happens to other threads in your program when the CLI steps your thread. For more information, see "Stepping" on page 59.

To save a P/T set definition for later use, assign the specifiers to a Tcl variable. For example:

```
set myset { g2.3 t3.1 }
dfocus $myset dgo
```

The thread of interest can also be modified by a *width* specifier. As the **dfocus** command returns its focus set, you can save this value for later use. For example:

```
set save_set [dfocus]
```

Examples

Here are some example specifiers:

- **d1.<** Use the default set for each command, focusing on the first user thread in process 1. The "<: sets the TID to the first user thread.
- **g2.3** Select process 2, thread 3 and set the width to "group".
- **t1.7** Commands act only on thread 7 of process 1.

You can leave out parts of the P/T set if what you enter is unambiguous. A missing width or PID is filled in from the current focus. A missing TID is always assumed to be <. For more information, see "Incomplete Arena Specifiers" on page 57.

Setting Group Focus

When you start a multiprocess program, the CLI adds each process to a control group as the process starts. TotalView decides which group it should place the process based on the system call—**fork()** or **execve()**—that created or changed the processes.

TotalView has two kinds of groups: process groups and thread groups. Process groups only contain processes. Thread groups only contain threads, but threads in a thread group can come from any process.

There are two different types of process groups:

- **Control Group**

  Contains the parent process and all related processes. A control group includes children that were forked (processes that share the same

source code as the parent) and children that were forked but which subsequently called **execve()**.

Assigning a new value to the **CGROUP(dpid)** variable for a process changes that processes control group. In addition, the **dgroups –add** command lets you add members to a group.

- **Share Group**

  Contains all members of a control group that share the same executable image. (Note, however, that dynamically loaded libraries may vary between share group members.)

  TotalView automatically places processes in share groups based on their control group and their executable image.

  NOTE You cannot change a share group's members.

In addition, there are also two types of thread groups:

- **Workers Group**

  Contains all worker threads from all processes in the control group. By default, it contains all threads except the kernel-level manager threads that can be identified. TotalView does not let you delete a workers group.

- **Lockstep Group**

  Contains every stopped thread in a share group that has the same PC. There is one lockstep group for every thread. For example, suppose two threads are stopped at the same PC. TotalView will create two lockstep groups. While each lockstep group has the same two members, they differ in that each has a different thread of interest.

  The group ID's value for a lockstep group differs from the ID of other groups. Rather than an automatically allocated integer ID, the lockstep group ID has the form **pid.tid**, where **pid.tid** identifies the thread with which it is associated. For example, the lockstep group for thread 2 in process 1 is **1.2**.

In general, if you are debugging a multiprocess program, the control group and share group differ only when the program has children that are forked with a call to **execve()**.

Specifying Groups in P/T Sets

The arena syntax that has so far been presented does not let you change the focus to a different group. This section shows how to add group specifiers when setting a focus.

If you do not include a group specifier, the default is the control group. The CLI only displays a target group in the focus string if you set it to something other than the default value.

NOTE Target group specifiers are most often used with the single-step commands as they give these commands more control over what is being stepped.

Here is how you add groups to the way you specify arenas:

[*width_letter*][*group_indicator*][PID][.PID]

This format adds the *group_indicator* to the previously discussed syntax. There are actually several different ways for specifying a group.

- You can name one of TotalView's predefined sets. These sets are identified by letters. For example, the following command sets the focus to the workers group:

  dfocus W

- You can identify a group by its number. For example, here is how you set the focus to group 3:

  dfocus 3/

  Notice the trailing slash. This slash lets the CLI know that you are specifying a group number instead of a PID. In contrast, here is an example that names process 3 within group 3:

  dfocus 3/3

- As you can also create named sets, you must surround these set names with slashes. For example, here is how you would use the **my_group** set of threads within process 3:

  dfocus p/my_group/3

In the formal description of this syntax, everything appears to be optional. While no single element is required, you must enter at least one element. TotalView will determine other values based on the current focus. When you specify a group, you can use either a *group_letter*, a *group_number*, or a *group_name*.

The *group_number* is a value that TotalView assigns to the group.

The slash character is optional if you are using a *group_letter*. However, you must use it as a separator when entering a numeric group ID and a **pid.tid** pair. For example, the following entry indicates process 2 in group 3:

```
p3/2
```

If you have created your own group, you need to place the name of this group within slash characters. For example, here is how you would indicate the set of threads within process 2 that are also within a named group:

```
p/my_group/2
```

The *group_letter* can be:

C *Control group*

All processes in the control group.

D *Default control group*

All processes in the control group. The only difference between this specifier and the **C** specifier is that **D** tells the CLI that it should not display a group letter within the CLI prompt.

S *Share group*

The set of processes in the control group that have the same executable as the arena's thread of interest.

W *Workers group*

The set of all worker threads in the control group.

L *Lockstep group*

A set containing all threads in the share group that have the same PC as the arena's thread of interest. If these threads are stepped as a group, they will proceed in lockstep.

The group letter is always in uppercase.

On some systems, TotalView cannot distinguish manager threads from user threads, so manager threads might be included by mistake. This means that you may need to remove these manager threads from workers groups by using the **dgroups –remove** or **dworker** commands.

Specifier Combinations

The following table indicates what specifier combinations mean for the CLI's stepping commands:

TABLE 2: Specifier Combinations

Specifier	Meaning
aC	Specifies all threads.
aS	Specifies all threads.
aW	Specifies all threads in all workers groups.
aL	Specifies all threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads".
gC	Specifies all threads in the thread of interest's control group.
gS	Specifies all threads in the thread of interest's share group.
gW	Specifies all worker threads in the control group containing the thread of interest.
gL	Specifies all threads in the same share group within the process containing the thread of interest that have the same PC.
pC	Specifies all threads in the control group of the process of interest. This is the same as **gC**.
pS	Specifies all threads in the process that participate in the same share group as the thread of interest.
pW	Specifies all worker threads in the process of interest.
pL	Specifies all threads in the process of interest whose PC is the same as the thread of interest.
tC tS tW tL	These four combinations, while syntactically correct, are meaningless. The **t** specifier overrides the group specifier. So, for example, **tW** and **t** both name the current thread.

NOTE Stepping commands behave differently if the group being stepped is a process group or a thread group. For example, "aC" and "aS" perform the same action while "aL" is different.

Here are a few examples:

pW3	All worker threads in process 3.
pW3.<	All worker threads in process 3. Notice that the focus of this specifier is the same as the previous example's.
gW3	All worker threads in the control group containing process 3. Notice the difference between this and **pW3**, which restricts the focus to one of the processes in the control group.
gL3.2	All threads in the same share group as process 3 that are executing at the same PC as thread 2 in process 3. The reason this is a share group and not a control group is that different share group can only reside within one control group.
/3	Specifies processes and threads in process 3. As the arena width, process of interest, and thread of interest are inherited from the existing P/T set, the exact meaning of this specifier depends on the previous context. While the "/" is unnecessary because no group is indicated, it is syntactically correct.
g3.2/3	The 3.2 group ID is the name of the lockstep group for thread 3.2. This group includes all threads in process 3's share group that are executing at the same PC as thread 2.
p3/3	Sets the process to process 3. The group of interest is set to group 3. If group 3 is a process group, most commands ignore the group setting. If group 3 is a thread group, most commands act upon all threads in process 3 that are also in group 3. Setting the process with an explicit group should be done with care as the combination may not be what you expect given that commands, depending on their scope, must look at the thread of interest, process of interest, and group of interest.

NOTE Specifying thread width with an explicit group ID probably does not mean much.

In the following examples, the first argument to the **dfocus** command defines a temporary P/T set upon which the CLI command (the last term) will operate. The **dstatus** command lists information about processes and threads.

NOTE The examples assume that the global focus was "d1.<" initially.

dfocus g dstatus
Displays the status of all threads in the control group.

dfocus gW dstatus
Displays the status of all worker threads in the control group.

dfocus p dstatus
Displays the status of all worker threads in the current focus process. The width here, as in the previous example, is process and the (default) group is the control group; intersecting this width and the default group creates a focus that is also the same as the previous example.

dfocus pW dstatus
Displays the status of all worker threads in the current focus process. The width is process level and the target is the workers group.

The following example shows how the prompt changes as you change the focus. Notice how the prompt changes when using the **C** and the **D** group specifiers.

```
d1.<> f C
dC1.<
dC1.<> f D
d1.<
d1.<>
```

All Does Not Always Mean *All*

When you use one of the stepping commands, TotalView determines the scope of what runs and what stops by looking at the thread of interest.

This section looks at the differences in behavior when the **a** (all) arena is used. Here is what runs when you use this arena:

TABLE 3: **a (all) Specifier Combinations**

Specifier	Meaning
aC	Specifies all threads.
aS	Specifies all threads.
aW	Specifies all threads in all workers groups.
aL	Specifies all threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads".

Here is what some combinations mean:

f aC dgo — Runs everything. If you are using the **g** command, everything after the **a** is ignored: **a/aPizza/17.2**, **ac**, **aS**, and **aL** do the same thing. TotalView runs everything.

f aC duntil — While everything runs, TotalView must wait until something reaches a goal. It may not be obvious what this *thing* is. Since **C** is a process group, all processes run until at least one thread in every participating process has arrived at a goal.

Since this goal must reside within the current share group, the command completes as soon as all processes in the thread of interest's share group have at least one thread at the goal.

Notice that the thread of interest determines the goal. If there are other control groups, they do not participate in the goal.

f aS duntil — This command performs identically to **f aC until** because, as was just mentioned, the goal for **f aC until** and **f aS until** are the same and the processes that are in this scope are identical.

Notice that there could be more than one share group within a control group. However, these other share groups do not participate in the goal.

f aL duntil	While everything will run, it is again not clear what should occur. **L** is a thread group, so you might expect that the **duntil** command will wait until all participating threads arrive at the goal. TotalView defines this set of threads as just those thread in the TOI's lockstep group. While there are other lockstep groups, these lockstep groups do not participate in the goal.
f aW duntil	While everything will run, TotalView will wait until all members of the thread of interest's workers group arrive at the goal.

There are two fundamental distinctions between process group and thread group behavior:

- When focus is on a process group, TotalView waits until just one thread from each participating process arrives at the goal. The other threads just run and TotalView does not care where they end up.

  When focus is on a thread group, every participating thread must arrive at the goal.
- When the focus is on a process group, TotalView steps a thread over the goal breakpoint and continues the process if it isn't the "right thread".

  When the focus is on a thread group, TotalView holds a thread even if it isn't the right thread. It also continues the rest of the process. This behavior only exists on systems that allow TotalView to have asynchronous thread control.

With this in mind, **f aL dstep** does not step all threads. Instead, it steps only the threads in the TOI's lockstep group. All other threads run freely until the stepping process for these lockstep threads is occurring.

Setting Groups

This section presents a series of examples that set and create groups. Many of the examples use CLI commands that have not yet been introduced. You will probably need to refer to the command's definition in Chapter 6 before you can appreciate what is occurring.

Here are some methods for indicating that thread 3 in process 2 is a worker thread.

dset WGROUP(2.3) $WGROUP(2)
Assigns the group ID of the thread group of worker threads associated with process 2 to the **WGROUP** variable. (Assigning a nonzero value to **WGROUP** indicates that this is a worker group.)

dset WGROUP(2.3) 1
A simpler way of doing the same thing as the previous example.

dfocus 2.3 dworker 1
Adds the groups in the indicated focus to a workers group.

dset CGROUP(2) $CGROUP(1)
dgroups -add -g $CGROUP(1) 2
dfocus 1 dgroups -add 2
These three commands insert process 2 into the same control group as process 1.

dgroups -add -g $WGROUP(2) 2.3
Adds process 2, thread 3 to the workers group associated with process 2.

dfocus tW2.3 dgroups -add
A simpler way of doing the same thing as the previous example.

Here are some additional examples:

dfocus g1 dgroups -add -new thread
Creates a new thread group that contains all the threads in all the processes in the control group associated with process 1.

set mygroup [dgroups -add -new thread $GROUP($SGROUP(2))]
dgroups -remove -g $mygroup 2.3
dfocus g$mygroup/2 dgo
Defines a new group containing all the threads in process 2's share group except for thread 2.3 and then continues that set of threads. The first command creates a new group containing all the threads from the share group, the second removes thread 2.3, and the third runs the remaining threads.

An Extended Example

The **g** specifier can sometimes be confusing when is coupled with a group. In many cases, the reason that you use **g** is to force the group when the current default focus indicates another kind of focus. Stated in another way, isn't something like **gL** redundant?

The following example will clarify why and when you use the **g** specifier. The first step is to set a breakpoint in a multithreaded OMP program and execute the program until it hits the breakpoint:

```
d1.<> dbreak 35
Loaded OpenMP support library libguidedb_3_8.so :
                              KAP/Pro Toolset 3.8
1
d1.<> dcont
Thread 1.1 has appeared
Created process 1/37258, named "tx_omp_guide_llnl1"
Thread 1.1 has exited
Thread 1.1 has appeared
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 1.1 hit breakpoint 1 at line 35 in ".breakpoint_here"
```

The default focus is **d1.<**, which means that the CLI is at its default width, The process of interest is 1, and the thread of interest is the lowest numbered nonmanager thread. Because the default width for the **dstatus** command is "process", entering **dstatus** tells the CLI to display the status of all processes. Notice that typing **dfocus p st** produces the same output:

```
d1.<> dstatus
1:       37258     Breakpoint     [tx_omp_guide_llnl1]
    1.1: 37258.1   Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
    1.2: 37258.2   Stopped        PC=0xffffffffffffffff
    1.3: 37258.3   Stopped        PC=0xd042c944
d1.<> dfocus p dstatus
1:       37258     Breakpoint     [tx_omp_guide_llnl1]
    1.1: 37258.1   Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
    1.2: 37258.2   Stopped        PC=0xffffffffffffffff
    1.3: 37258.3   Stopped        PC=0xd042c944
```

Here's what the CLI displays when you ask for the status of the lockstep group. (The rest of this example will use the **f** abbreviation for **dfocus**.)

```
d1.<> f L dstatus
1:       37258     Breakpoint     [tx_omp_guide_llnl1]
   1.1: 37258.1    Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
```

This command tells the CLI to get the status of the threads in thread 1.1's (the thread of interest) lockstep group. The **f L** command modifier narrows the set so that the display only includes the threads in the process that are at the same PC as the thread of interest.

NOTE By default, the dstatus command displays information at "process" width. This means that you do not need to type "f pL dstatus".

The next command runs thread 1.3 to the same line as thread 1.1. This is immediately followed by a command that displays the status of all the threads in the process:

```
d1.<> f t1.3 duntil 35
   35@>      write(*,*)"i= ",i,
                  "thread= ",omp_get_thread_num()
d1.<> f p dstatus
1:       37258     Breakpoint     [tx_omp_guide_llnl1]
   1.1: 37258.1    Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
   1.2: 37258.2    Stopped        PC=0xffffffffffffffff
   1.3: 37258.3    Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
```

As expected, the CLI has added a thread to the lockstep group:

```
d1.<> f L dstatus
1:       37258     Breakpoint     [tx_omp_guide_llnl1]
   1.1: 37258.1    Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
   1.3: 37258.3    Breakpoint     PC=0x1000acd0,
                                      [./tx_omp_llnl1.f#35]
```

The next set of commands first narrows the width of the default focus to thread width—notice that the prompt changes—then displays the contents of the lockstep group.

```
d1.<> f t
t1.<> f L dstatus
1:        37258     Breakpoint     [tx_omp_guide_llnl1]
    1.1:  37258.1   Breakpoint     PC=0x1000acd0,
                                       [./tx_omp_llnl1.f#35]
```

This is the hard step. While the lockstep group of the thread of interest has two threads, the current focus has only thread 1, and that thread is, of course, part of the lockstep group. Consequently, the lockstep group *in the current focus* is just the one thread.

If you ask for a wider width (**p** or **g**) with **L**, the CLI displays more threads from the lockstep group of thread 1.1.

```
t1.<> f pL dstatus
1:        37258     Breakpoint     [tx_omp_guide_llnl1]
    1.1:  37258.1   Breakpoint     PC=0x1000acd0,
                                       [./tx_omp_llnl1.f#35]
    1.3:  37258.3   Breakpoint     PC=0x1000acd0,
                                       [./tx_omp_llnl1.f#35]
t1.<> f gL dstatus
1:        37258     Breakpoint     [tx_omp_guide_llnl1]
    1.1:  37258.1   Breakpoint     PC=0x1000acd0,
                                       [./tx_omp_llnl1.f#35]
    1.3:  37258.3   Breakpoint     PC=0x1000acd0,
                                       [./tx_omp_llnl1.f#35]
t1.<>
```

NOTE If the thread of interest is 1.1, "L" refers to group number 1.1, which is the lockstep group of thread 1.1.

Because this example only contains one process, the **pL** and **gL** modifiers produce the same result when used with **dstatus**. If, however, the program had additional processes in the group, you could only see them by using a **gL** modifier.

In this example, the focus indicated by the prompt—this focus is called the *outer* focus—controls the display. Notice what happens when **dfocus** commands are strung together:

```
t1.<> f d
d1.<
d1.<> f tL dstatus
```

```
1:          37258     Breakpoint     [tx_omp_guide_lnl1]
    1.1:    37258.1   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]

d1.<> f tL f p dstatus
1:          37258     Breakpoint     [tx_omp_guide_lnl1]
    1.1:    37258.1   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]
    1.3:    37258.3   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]

d1.<> f tL f p f D dstatus
1:          37258     Breakpoint     [tx_omp_guide_lnl1]
    1.1:    37258.1   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]
    1.2:    37258.2   Stopped        PC=0xffffffffffffffff
    1.3:    37258.3   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]

d1.<> f tL f p f D f L dstatus
1:          37258     Breakpoint     [tx_omp_guide_lnl1]
    1.1:    37258.1   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]
    1.3:    37258.3   Breakpoint     PC=0x1000acd0,
                                         [./tx_omp_lnl1.f#35]

d1.<>
```

Stringing multiple focuses together may not produce the most readable result, and this example illustrates how one **dfocus** command can modify what another sees and will act upon. The ultimate result is an arena upon which a command will act. In these examples, the **dfocus** command is telling the **dstatus** command what it should be displaying.

Incomplete Arena Specifiers

In general, you do not need to completely specify an arena. Missing components are assigned default values or are filled in from the current focus. The only requirement is that the meaning of each part of the specifier cannot be ambiguous. Here is how the CLI fills in missing pieces:

- If you do not use a width, the CLI uses the width from the current focus.
- If you do not use a PID, the CLI uses the PID from the current focus.

- If you set the focus to a list, there is no longer a default arena. This means that you must explicitly name a width and a PID. You can, however, omit the TID. (If you omit the TID, the CLI defaults to <.)

  You can type a PID without typing a TID. If you omit the TID, the CLI uses its default of "<", where "<" indicates the lowest numbered worker thread in the process. If, however, the arena explicitly names a thread group, < means the lowest numbered member of the thread group.

  The CLI does not use the TID from the current focus, since the TID is a process-relative value.
- A dot typed before or after the number lets the CLI know if you are specifying a process or a thread. For example, "**1.**" is clearly a PID, while "**.7**" is clearly a TID.

  If you type a number without typing a period, the CLI interprets the number as being a PID.
- If the width is **t**, you can omit the dot. For instance, **t7** refers to thread 7.
- If you enter a width and do not specify a PID or TID, the CLI uses the PID and TID from the current focus.

  If you use a letter as a group specifier, the CLI obtains the rest of the arena specifier from the default focus.
- You can use a group ID or tag followed by a "/". The CLI obtains the rest of the arena from the default focus.

Lists with Inconsistent Widths

The CLI lets you create lists containing more than one width specifier. While this can be very useful, it can be confusing. Consider the following:

```
{ p2 t7 g3.4 }
```

This list being defined is quite explicit: all of process 2, thread 7, and all processes in the same group as process 3, thread 4. However, how should the CLI use this set of processes, groups, and threads?

In most cases, the CLI does what you would expect it to do: a command iterates over the list and acts on each arena. If the CLI cannot interpret an inconsistent focus, it prints an error message.

There are commands that act differently. These commands use each arena's width to determine the number of threads on which it will act. This

is exactly what the **dgo** command does. In contrast, the **dwhere** command creates a call graph for process-level arenas, and the **dstep** command runs all threads in the arena while stepping the thread of interest. It may wait for threads in multiple processes for group-level arenas.

Stepping

The action that the CLI will perform when stepping assembler instructions or source statements depends on whether you have specified thread, process, or group width.

NOTE If you do not explicitly name a group, the CLI steps the control group.

The following sections describe what happens at each width. In all cases, if a thread hits an action point other than the goal breakpoint during a stepping operation, the operation ends.

Thread

TotalView steps the thread of interest (TOI). Stepping a thread is not the same as stepping a thread's process because a process can have more than one thread.

NOTE Thread stepping is not implemented on Sun platforms. On SGI platforms, thread stepping is not available with pthread programs. If, however, your program's parallelism is based on SGI's sprocs, thread stepping is available.

Stepping at thread width (**t**) tells the CLI that it should just run that thread. In contrast, process width (**p**) tells the CLI that it should run all threads in the process that are allowed to run while the thread of interest is stepped.

TotalView also allows all manager threads to run freely while the stepping action is occurring.

Process

The behavior (which is the default) depends on whether the group of interest (GOI) is set to a process group or a thread group. If the GOI is a:

- P*rocess group*, TotalView runs all threads in the process, and execution continues until the thread of interest arrives at its goal location, which can be the next statement, the next instruction, and so on. Only when the TOI reaches the goal are the other threads in the process stopped.
- T*hread group*, the behavior differs. All threads in the GOI, and all manager threads are allowed to run. As each member of the GOI arrives at the goal, it is stopped; the rest of the threads are allowed to continue. The command finishes when all members of the GOI arrive at the goal. At that point, TotalView stops the whole process.

Group

The behavior again depends on whether the GOI is a process group or a thread group. If the GOI is a:

- P*rocess group*, TotalView examines that group and identifies each process in it which has a thread stopped at the same location as the thread of interest (a *matching* process). TotalView runs all processes in the control group associated with the process of interest. Each time a thread arrives at the goal, the process containing that thread is stopped. The command finishes after TotalView stops all "matching" processes. At that time, all members of the control group will also be stopped.
- T*hread group*, TotalView again runs all processes in the control group. However, as each thread arrives at the goal, TotalView just stops that thread; the rest of the threads in the process containing it are allowed to continue. The command finishes when all threads in the group of interest have arrived at the goal. (TotalView does not wait for threads that are not in the same share group as the thread of interest since they are executing different code and can never arrive at the goal.) When the command finishes, TotalView again stops all process in the control group.

Using duntil

The **duntil** command differs from other step commands when you apply it to a process group. (The **duntil** command tells TotalView to execute program statements *until* a selected statement is reached.) When applied to a process group, it identifies all processes in the group already having a thread stopped at the goal. These are the *matching* processes. It then runs only the nonmatching processes. Whenever a thread arrives at the goal,

TotalView stops its process. The command finishes when all members of the group are stopped. This lets you *sync up* all the processes in a group in preparation for group-stepping them.

In all cases, if a process does not exist before a command executes, TotalView creates it before executing the command.

How do I ...

Here are some of the operations that can occur when you are using the CLI's stepping commands:

- **Step a single thread**

  While the thread runs, no other thread runs (except kernel manager threads).

  Example: dfocus t dstep

- **Step a single thread while the process runs**

  A single thread runs into or through a critical region.

  Example: dfocus p dstep

- **Step one thread in each process in the group**

  While one thread in each process in the share group runs to a goal, the rest of the threads run freely.

  Example: dfocus g dstep

- **Step all worker threads in the process while nonworker threads run**

  Runs worker threads through a parallel region in lockstep.

  Example: dfocus pW dstep

- **Step all workers in the share group**

  All processes in the share group participate. The nonworker threads run.

  Example: dfocus gW dstep

- **Step all threads that are at the same PC as the thread of interest**

  The CLI selects threads from one process or from the entire share group. This differs from the previous two bullets in that the CLI uses the set of threads that are in lockstep with the thread of interest rather than using the workers group.

  Example: dfocus L dstep

In the following examples, the default focus is set to **d1.<**.

dstep	Steps the thread of interest while running all other threads in the process.
dfocus W dnext	Runs the thread of interest and all other worker threads in the process to the next statement. Other threads in the process run freely.
dfocus W duntil 37	Runs all worker threads in the process to line 37.
dfocus L dnext	Runs the thread of interest and all other stopped threads at the same PC to the next statement. Other threads in the process run freely. Threads that encounter a temporary breakpoint in the course of hopping to the next statement usually join the lockstep group.
dfocus gW duntil 37	Runs all worker threads in the share group to line 37. Other threads in the control group run freely.
UNW 37	Performs the same action as the previous command: runs all worker threads in the share group to line 37. This example uses the predefined **UNW** alias instead of the individual commands. That is, **UNW** is an alias for **dfocus gW duntil**.
SL	Finds all threads in the share group that are at the same PC as the thread of interest and steps them all one statement. This command is the built-in alias for **dfocus gL dstep**.
sl	Finds all threads in the current process that are at the same PC as the thread of interest, and steps them all one statement. This command is the built-in alias for **dfocus L dstep**.

"Piling Up" vs. "Running Through"

If you use a step command when the focus is set to the lockstep group, the CLI runs all threads at the same PC as the thread of interest to the same goal. In contrast, when you enter a step command with its focus set to a

nonlockstep thread group, the CLI runs all threads in the indicated group to the same goal as the thread of interest. The set of threads being run to the goal is called the *active set*. In all cases, TotalView selects the goal based on the PC of the TOI. The step command tells TotalView that it must wait for all threads in the active set to reach the goal.

While the active set is running to the goal, all other threads in the process (or group) run freely.

If a thread that is not in the active set reaches the goal breakpoint, the CLI continues that process again until all active threads reach the goal. Before continuing the process, the CLI can either step the thread over the goal breakpoint, which allows it to run through, or it can hold it, thus causing threads to pile up at the goal.

If a single thread is being run into a critical region, threads that are not in the active set run freely. Otherwise, the thread of interest may not be able to make any progress.

If a lockstep group is running, threads that are not in the active set pile up at the goal when the CLI steps the lockstep group.

NOTE Threads pile up when a thread group is specified, and they "run through" when a process group is specified.

"Piling up" can only occur for systems in which the CLI can control threads asynchronous.

P/T Set Expressions

At times, you do not want all of one kind of group or process to be in the focus set. The CLI lets you use the following three operators to manage your P/T sets:

|	Creates a union; that is, all members of the sets.
-	Creates a difference; that is, all members of the first set that are not also members of a second set.
&	Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, here is how you would create a union of two P/T sets:

```
p3 | L2
```

A set operator only operates upon two sets. You can, however, apply these operations repeatedly. For example:

```
p2 | p3 & L2
```

This statement creates a union between **p2** and **p3**, and then creates an intersection between the union and **L2**.

The CLI associates sets from left to right. You can change this order by using parentheses. For example:

```
p2 | (p3 & pL2)
```

Typically, these three operators are used with the following P/T set functions:

breakpoint()	Returns a list of all threads that are stopped at a breakpoint.
error()	Returns a list of all threads stopped due to an error.
existent()	Returns a list of all threads.
held()	Returns a list of all threads that are held.
nonexistent()	Returns a list of all processes that have exited or which, while loaded, have not yet been created.
running()	Returns a list of all running threads.
stopped()	Returns a list of all stopped threads.
unheld()	Returns a list of all threads that are not held.
watchpoint()	Returns a list of all threads that are stopped at a watchpoint.

The argument that all of these operators use is a P/T set. You specify this set in the same that a P/T set is specified for the **dfocus** command. For example, **watchpoint(L)** returns all threads in the current lockstep group.

The following examples should clarify how these operators and functions are used. The P/T set that is the argument to these operators is **a** (all).

f {breakpoint(a) | watchpoint(a)} dstatus
: Shows all threads that stopped at breakpoints and watchpoints. The **a** argument is the standard P/T set indicator for "all".

f { stopped(a) - breakpoint(a) } dstatus
: Shows all stopped threads that are not stopped at breakpoints.

f { g.3 - p6 } duntil 577
: Runs thread 3 along with all other processes in the group to line 577. However, do not run anything in process 6.

f { ($PTSET) & p123 }
: Uses just process 123 within the current P/T set.

Chapter 4

Using the CLI

The two components of the Command Line Interface (CLI) are the Tcl-based programming environment and the commands added to the Tcl interpreter that allow you to debug your program. This chapter looks at how these components interact and describes how you specify processes, groups, and threads.

This chapter tends to emphasize interactive use of the CLI rather than using the CLI as a programming language because many of the concepts that will be discussed are easier to understand in an interactive framework. However, everything in this chapter can be used in both environments.

How a Debugger Operates

The CLI and TotalView debuggers affect the program but are not part of the program's process. That is, TotalView and the CLI run in separate processes, and their semantics are separate from the program's semantics.

A CLI interaction has two kinds of input: the executables that make up the program *and* the Tcl and CLI commands that you type. You will prepare the executable for debugging by compiling it with the **–g**, which tells the compiler to add information that lets the CLI display high-level output to the user, expressed in terms of the procedures and variables used in the source code. This option also allows the CLI to access components of the program (such as source files), eliminating the need for some assistance from the user.

If you do not use **–g** option, TotalView only displays the assembler code generated by the compiler.

Tcl and the CLI

The TotalView CLI is built within version 8.0 of Tcl, which means that TotalView's CLI commands are built into Tcl. The CLI is not a library of commands that you can bring into other implementations of Tcl. Because the Tcl you are running is the standard 8.0 version, the TotalView CLI supports all libraries and operations that run using version 8.0 of Tcl.

Integrating CLI commands into Tcl makes them intrinsic Tcl commands. This means that you can enter and execute CLI commands in exactly the same way as you enter and execute built-in Tcl functions such as **file** or **array**. It also means that you can embed Tcl primitives and functions within CLI commands.

For example, you can create a Tcl list that contains a list of threads, use Tcl commands to manipulate that list, and then have a CLI command operate on the elements of this list. Or, you create a Tcl function that dynamically builds the arguments that a process will use when it begins executing.

Because the CLI is an integral part of TotalView's version of Tcl, there are no differences between using a CLI command and using a Tcl command. Furthermore, all CLI operations can be manipulated by Tcl.

The CLI and TotalView

The following figure illustrates the relationship between the CLI, the TotalView GUI, the TotalView core, and your program:

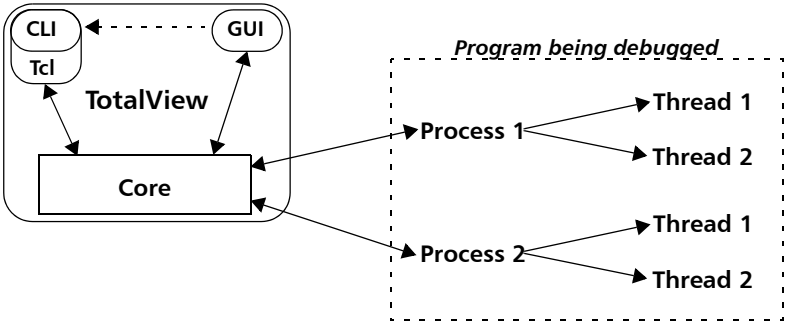

FIGURE 23: **The CLI and TotalView**

The CLI and the GUI are interfaces that communicate with the TotalView core, which is the component that actually performs the debugging work. In this figure, the dotted arrow between the GUI and the CLI indicates that you can invoke the CLI from the GUI. The reverse is not true: you cannot invoke the GUI from the CLI.

In turn, the TotalView core communicates with the processes that make up your program and receives information back from these processes, and passes them back to the component that sent the request.

The CLI Interface

The way in which you interact with the CLI is by entering a CLI command. Typically, the effect of executing a CLI command is one or more of the following:

- The CLI displays information about the program.
- A change takes place in the program's state.
- A change takes place in the information that the CLI maintains about the program.

The CLI signals that it has completed a command by displaying a prompt.

Although CLI commands are executed sequentially, commands executed by your program may not be. For example, the CLI does not require that your program be stopped when it prompts for and performs commands. It only requires that the last CLI command be complete before it can begin executing the next one. In many cases, the processes and threads being debugged continue to execute while the CLI is performing commands.

Because actions are occurring constantly, state information displayed by the CLI is usually mixed in with the commands that you type.

Entering Ctrl-C while a CLI command is executing interrupts that CLI command or executing Tcl macro. If the CLI is displaying its prompt, typing Ctrl-C stops executing processes.

Starting the CLI

You can start the CLI in two ways:

- You can start the CLI from within the TotalView window by selecting the **Tools** > **Command Line** command within the Root and Process windows. After selecting this command, TotalView opens a window into which you can enter CLI commands.
- You can start the CLI directly from a shell prompt by typing **totalviewcli**. (This assumes that the TotalView binary directory is in your path.)

Here is a snapshot of a CLI window that shows part of a program being debugged:

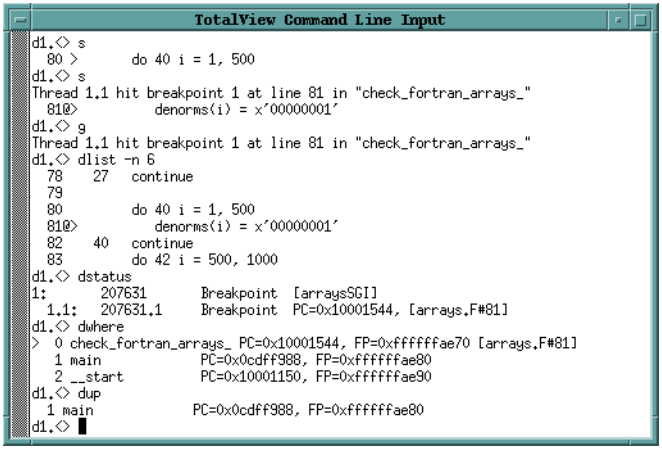

FIGURE 24: **CLI xterm Window**

If you have problems entering and editing commands, it could be because you invoked the CLI from a shell or process that manipulates your **stty** settings. You can eliminate these problems if you use the **stty sane** CLI command. (If the **sane** option is not available, you will have to change values individually.)

If you start the CLI using the totalviewcli command, you can use all of the command line options that you can use when starting TotalView. For more information, see the TOTALVIEW USERS GUIDE.

Initializing the Debugger

An *initialization file* contains commands that let you modify the TotalView and CLI environments and add your own functions to this environment. TotalView allows you to place information in more than one file. These files can be located in your installation directory, your home directory, or the directory from which you invoked TotalView. If it is present in one of these places, TotalView reads and executes its contents.

Typically, **.tvdrc** files contain command, function, variable definitions, and function calls that you want executed whenever you start a new debugging session.

If you add the **-s** *filename* option to either the **totalview** or **totalviewcli** shell commands, you can have TotalView execute the CLI commands contained within *filename*. Your startup file executes after **.tvdrc** files execute.

The following figure shows the order in which initialization and startup files execute:

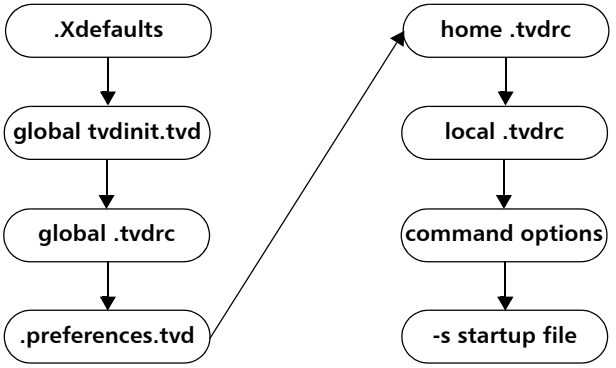

FIGURE 25: **Startup and Initialization Sequence**

The **-s** option lets you, for example, initialize the debugging state of your program, run the program you are debugging until it reaches some point where you are ready to begin debugging, and even lets you create a shell command that starts the CLI.

NOTE The .Xdefaults file, which is actually read by the server when you start X Windows, is only used by the GUI. The CLI ignores it.

As part of the initialization process, TotalView exports two environment variables into your enviroment: **LM_LICENSE_FILE** and either **SHLIB_PATH** or **LD_LIBRARY_PATH**.

If you have saved a breakpoint file into the same subdirectory as your program, TotalView automatically reads the information in this file when it loads your program.

NOTE The format of a Release 5.0 breakpoint file differs from that used in earlier releases. While Release 5 versions of TotalView can read breakpoint files created by earlier versions, earlier versions cannot read a Release 5 breakpoint file.

You can also invoke scripts by naming them in the **TV::process_load_callbacks** list. For information, see "Initializing TotalView After Loading an Image" on page 99.

Start-up Example

Here is a very small CLI script:

```
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT { 0 4 -wp}
dstep
catch { make_actions fork_loop.cxx } msg
puts $msg
```

This script begins by loading and interpreting the **make_actions.tcl** file, which was described in Chapter 2. It then loads the **fork_loop** executable, sets its default startup arguments, then steps one source-level statement.

If this were stored in a file named **fork_loop.tvd**, here is how you would tell TotalView to start the CLI and execute this file:

```
totalviewcli -s fork_loop.tvd
```

Information on options and X resources can be found in the TOTALVIEW USER'S GUIDE.

The following example shows how you would place a similar set of commands in a file that you would invoke from the shell:

```
#!/bin/sh
# Next line executed by shell, but ignored by Tcl because of: \
      exec totalviewcli -s "$0" "$@"
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

Notice that the only difference is the first few lines in the file. In the second line, the continuation character is ignored by the shell. It is, however, processed by Tcl. This means that the shell will execute the command while Tcl will ignore it.

Starting Your Program

The CLI lets you start debugging operations in several ways. To execute your program from within the CLI, enter a **dload** command followed by the **drun** command. The following example uses the **totalviewcli** command to start the CLI. This is followed by **dload** and **drun** commands. As this was not the first time the file was run, breakpoints exist from a previous session.

NOTE In this listing, the CLI prompt is "d1.<>". The information preceding the ">" symbol indicates the processes and threads upon which the current command acts. The prompt is discussed in "Command and Prompt Formats" on page 81.

```
% totalviewcli
IRIX6 MIPS TotalView 5X.0.0-7
Copyright 1999-2000 by Etnus, LLC. ALL RIGHTS RESERVED.
Copyright 1999 by Etnus, Inc.
Copyright 1996-1998 by Dolphin Interconnect Solutions, Inc.
Copyright 1989-1996 by BBN Inc.

tcl_library is set to "/opt/totalview/lib"
```

```
d1.<> dload arrays      # load the "arrays" program
Mapping 430 bytes of ELF string data from 'arrays'...done
Digesting 42 ELF symbols from 'arrays'...done
Skimming 1825 bytes of DWARF '.debug_info' symbols from
      'arrays'...done
Indexing 408 bytes of DWARF '.debug_frame' symbols from
      'arrays'...done
...
Loading 1122 bytes of DWARF '.debug_info' information for
      arrays.F...done
1
d1.<> dactions          # show action points
2 action points for process 1:
      1 addr=0x1000114c [arrays.F#53] Enabled
      2 addr=0x10000f34 [arrays.F#29] Enabled

d1.<> drun              # run "arrays" until first action point
Created process 1/10715, named "arrays"
Thread 1.1 has appeared
d1.<> Thread 1.1 hit breakpoint 2 at line 29 in
      "check_fortran_arrays_"
```

This two-step operation of loading then running allows you to set action points before execution begins. It also means that you can execute a program more than once, keeping TotalView state settings (such as the location of action points) in effect. At a later time, you can use the **drerun** command to tell the CLI to restart program execution, perhaps sending it new command-line arguments. In contrast, reentering the **dload** command tells the CLI to reload the program into memory (for example, after editing and recompiling the program). The **dload** command always creates new processes.

The **dkill** command terminates one or more processes of a program started by using **dload**, **drun**, or **drerun**. The following contrived example continues where the previous example left off:

```
d1.<> dkill             # kill process
Process 1 has exited
d1.<> drun              # runs "arrays" from beginning
Created process 1/10722, named "arrays"
Thread 1.1 has appeared
d1.<> Thread 1.1 hit breakpoint 2 at line 29 in
```

```
"check_fortran_arrays_"
dlist -e -n 3                #Shows lines about execution point
Loading 168 bytes of DWARF '.debug_info' information
for /comp2/mtibuild//targ64_m4/libftn/lseek64_.s...done
Loading 760 bytes of DWARF '.debug_info' information for
        vtan.c...done
Loading 1864 bytes of DWARF '.debug_info' information for
    ns_passwd.c...done
    28  do 10 i = 1, 100
    29@> master_array (i) = i * i * i
    30  0   continue
d1.<> dwhat master_array     # Show me information
Loading 901 bytes of DWARF '.debug_info' information for
        main.c...done
In thread 1.1:
Name: master_array; Type: integer*4(100); Size: 400 bytes;
        Addr: 0xfffffac90
    Address class: auto_var (Local variable)Loading 188 bytes of
    DWARF '.debug_info' information for
/xlv55/irix/lib/libc/libc_64_M4/csu/crt1tinit.s...done

d1.<> drun                   # Notice the error message
drun: Process 1 already exists. Kill it first, or use rerun.
d1.<> dkill                  # kill processes again
Process 1 has exited

d1.<> drun
Created process 1/10730, named "arrays"
Thread 1.1 has appeared
d1.<> Thread 1.1 hit breakpoint 2 at line 29 in
"check_fortran_arrays_"
```

Notice that messages from the CLI and TotalView are interleaved (sometimes inconveniently) throughout the interaction. The occurs because the CLI prompt lets you know that the CLI is ready to accept another command. In contrast, the CLI displays messages about your program's state when something happens within the executing process. This means that you are almost always going to have interactions like this.

NOTE You can minimize the amount of information that the CLI displays by setting the VERBOSE state variable to ERROR. See "dset" on page 200 for more information.

Because information is interleaved, you may not realize that the prompt has appeared. It is always safe to use the Enter key to have the CLI redisplay its prompt. If a prompt is not displayed after you press Enter, then you know that the CLI is still executing.

CLI Output

A CLI command can either print its output to a window, or it can return the output as a character string. If the CLI executes a command that returns a string value, it also prints the returned string. Most of the time, you are not concerned with the difference between *printing* and *returning-and-printing*. Either way, information is displayed in your window. And, in both cases, printed output is fed through a simple *more* processor. (This is discussed in more detail in the next section.)

Here are two cases where it matters whether output is printed directly or returned and then printed:

- When the Tcl interpreter executes a list of commands, only the information returned from the last command is printed. Information returned by other commands is not shown.
- You can only assign the output of a command to a variable if the command's output is returned by the command. Output that is printed directly cannot be assigned to a variable or otherwise manipulated unless you save it by using the **capture** command.

For example, the **dload** command returns the ID of the process object that was just created. The ID is normally printed—unless, of course, the **dload** command appears in the middle of a list of commands. For example:

```
{ dload test_program ; dstatus }
```

In this case, the CLI does not display the ID of the loaded program since **dload** was not the last command. On the other hand, you can easily assign the ID of the new process to a variable:

```
set pid [dload test_program]
```

In contrast, you cannot assign the output of the **help** command to a variable. For example, the following does not work:

```
set htext [help]
```

This statement assigns an empty string to **htext** because **help** does not return text; it just prints information.

To capture the output of a command that prints its output, use the **capture** command. For example, the following places the output of the **help** command into a variable:

```
set htext [capture help]
```

NOTE You can only capture the output from commands. You cannot capture the informational messages displayed by the CLI that describe process state.

"more" Processing

When the CLI displays output, it sends data through a simple internal *more*-like process. This process prevents data from scrolling off the screen before it can be viewed. After you see the **MORE** prompt, you must press Enter to continue with the next screen of data. If you type **q** (followed by a Enter), any remaining buffered output is discarded.

You can control the number of lines displayed between prompts by setting the **LINES_PER_SCREEN** state variable. (See **dset** for more information.)

Command Arguments

The default command arguments for a process are stored in the **ARGS(***num***)** variable, where **#** is the CLI ID for the process. If the **ARGS(***num***)** variable is not set for a process, the CLI uses the value stored in the **ARGS_DEFAULT** variable. **ARGS_DEFAULT** is set if you had used the **–a** option when starting the CLI or TotalView.

NOTE The –a option tells TotalView to pass the information that follows to the program.

For example:

```
totalviewcli -a argument-1, argument-2, ...
```

To set (or clear) the default arguments for a process, you can use **dset** to modify the **ARGS()** variables directly, or you can start the process with the **drun** command. For example, here is how you can clear the default argument list for process 2:

```
dunset ARGS(2)
```

The next time process 2 is started, the CLI uses the arguments contained within **ARGS_DEFAULT**.

You can also use the **dunset** command to clear the **ARGS_DEFAULT** variable. For example:

```
dunset ARGS_DEFAULT
```

All commands (except **drun**) that create a process—including **dgo**, **drerun**, **dcont**, **dstep**, and **dnext**—pass the default arguments to the new process. The **drun** command differs in that it replaces the default arguments for the process with the arguments that are passed to it.

Symbols

This section discusses how the CLI handles *symbols* and other names corresponding to various entities within your program or within TotalView.

Namespaces

CLI interactive commands exist within the primary Tcl namespace (::). Many TotalView state variables also reside in this namespace. However, the CLI and TotalView place functions and variables that are not ordinarily accessed in interactive sessions in other namespaces. These namespaces are:

TV::	Contains commands and variables that are most often used in scripts; that is, they are seldom used in an interactive debugging session.
TV::GUI::	Contains state variables that define and describe properties of the user interface such as window placement, color, and the like.

If you discover other namespaces beginning with **TV**, you have found a place containing internal functions and variables. These objects can (and will) change and disappear. So, don't use them. Also, do not create namespaces that begin with **TV** as you could cause problems by interfering with built-in functions and variables.

The CLI's **dset** command lets you set the value of these variables. You can ask the CLI to display a list of these variables by specifying the namespace. For example:

```
dset TV::
```

Symbol Names and Scope

Many commands refer to one or more program objects by using symbol names as arguments. In addition, some commands take expressions as arguments, where the expression can contain symbol names representing program variables.

NOTE Because the CLI is built on top of TotalView, the way in which the CLI interprets symbols is the way that TotalView interprets them.

TotalView learns about a program's symbols and their relationships by reading the debugging information that was generated when the program was compiled. This information includes a mapping from symbol names to descriptions of objects, providing information about a symbol's use (for example, a function), where it is located in memory after the executable is loaded, and associated features (for example, number and data types of a function's arguments). While TotalView smooths over many differences, the information provided by compiler manufacturers is not uniform, and differences exist between the kinds of information provided by Fortran, C, and C++ compilers.

In all cases, *scope* is central to the way TotalView interprets and accesses symbols. (A *scope* defines what part or how much of a program knows about about a symbol. For example, a variable that is defined with a subroutine is scoped to all statements within the subroutine. It is not scoped outside of the subroutine.) A program consists of one or more scopes that are estab-

lished by the program's structure. Typically, some scopes are nested within others. Every statement in a program is associated with a particular scope, and indirectly with the other scopes containing that scope.

Whenever a CLI command contains a symbol name, TotalView consults the program's symbol table to discover what object it refers to—this process is known as *symbol lookup*. As programming languages do not require that a symbol names be unique, determining which symbol it should use can be complicated. A symbol lookup is performed with respect to a particular context, expressed in terms of a single thread of execution. Each context uniquely identifies the scope to which a symbol name refers.

Qualifying Symbol Names

The syntax for qualifying a symbol with a scope closely resembles that for specifying a source location. The scopes within a program form a tree, with the outermost scope as the root. At the next level are executable files and dynamic libraries; further down are compilation units (source files), procedures, and other scoping units (for example, blocks) supported by the programming language. Qualifying a symbol is equivalent to specifying which scope it is in, or describing the path to a node in the tree. This is similar to describing the path to a file in a tree-structured file system.

A symbol is *fully qualified* in terms of its scope when all levels of the tree are included:

[#*executable-or-lib*#][*file*#][*procedure-or-line*#]*symbol*

In this definition, the pound sign (**#**) is a separator character.

TotalView interprets the components of the symbol name as follows:

- Just as file names need not be qualified with a full path, you can qualify a symbol's scope without including all levels in the tree.
- If a qualified symbol begins with **#**, the name that follows indicates the name of the executable or shared library (just as an absolute file path begins with a directory immediately within the root directory). If the executable or library component is omitted, the qualified symbol does not begin with **#**.

- The source file's name may appear after the (possibly omitted) executable or shared library.
- Because programming languages typically do not let you name blocks, that portion of the qualifier is specified as a line number within the block.
- The procedure name or block component (represented by a line number from that block) may appear after the (possibly omitted) source file name. This component is followed by **#**.
- The symbol name follows the (possibly omitted) procedure or block name. Since qualified symbols often appear in the context of an expression, the final symbol name could be followed by a dot (.), plus the name of a field from a class, union, or structure.

You can omit any part of the scope specification that is not needed to uniquely identify the symbol. Thus, **foo#x** identifies the symbol **x** in the procedure **foo**. In contrast, **#foo#x** identifies either procedure **x** in executable **foo** or variable **x** in a scope from that executable.

Similarly, **#foo#bar#x** identifies variable **x** in procedure **bar** in executable **foo**. If **bar** were not unique within that executable, the name would be ambiguous unless you further qualified it by providing a file name. Ambiguities can also occur if a file-level variable (common in C programs) has the same name as variables declared within functions in that file. For instance, **bar.c#x** refers to a file-level variable, but the name can be ambiguous when there are different definitions of **x** embedded in functions occurring in the same file. In this case, you would need to say **bar.c#1#x** to identify the scope that corresponds to the "outer level" of the file (that is, the scope containing line 1 of the file).

You can use the **dwhat** command to determine if an unqualified or partially qualified symbol name is ambiguous.

Command and Prompt Formats

The appearance of the CLI prompt lets you know that the CLI is ready to accept a command. This prompt lists the current focus, and then displays

a greater-than symbol (>) and a blank space. (The *current focus* is the processes and threads to which the next command applies.) For example:

d1.<>	The current focus is the default set for each command, focusing on the first user thread in process 1.
g2.3>	The current focus is process 2, thread 3; commands act on the entire group.
t1.7>	The current focus is thread 7 of process 1.
gW3.>	All worker threads in the control group containing process 3.
p3/3	Sets the process to process 3. The group of interest is set to group 3.

You can change the prompt's appearance by using the **dset** command to set the **PROMPT** state variable. For example:

```
dset PROMPT "Kill this bug! > "
```

Built-In Aliases and Group Aliases

Almost every CLI command has an alias that allows you to abbreviate the command's name. (An alias is one or more characters that the Tcl interprets as a command or command argument.)

NOTE The "alias" command (see Chapter 5) lets you create your own aliases.

After a few minutes of entering CLI commands, you will quickly come to the conclusion that it is much more convenient to use the command abbreviation. For example, you could type:

```
dfocus g dhalt
```

(This command tells the CLI to halt the current group.) It is much easier to type:

```
f g h
```

While less-used commands are often typed in full, a few commands are almost always abbreviated. These command include **dbreak** (**b**), **ddown** (**d**), **dfocus** (**f**), **dgo** (**g**), **dlist** (**l**), **dnext** (**n**), **dprint** (**p**), **dstep** (**s**), and **dup** (**u**).

The CLI also includes uppercase "group" versions of aliases for a number of commands, including all stepping commands. For example, the alias for **dstep** is "**s**"; in contrast, "**S**" is the alias for "**dfocus g dstep**". (The first command tells the CLI to step the process. The second steps the control group.)

Group aliases differ from the kind of group-level command that you would type in two ways:

- They do not work if the current focus is a list. The **g** focus specifier modifies the current focus, and it can only be applied if the focus contains just one term.
- They always act on the group, no matter what width is specified in the current focus. Therefore, **dfocus t S** does a step-group command.

Effects of Parallelism on TotalView and CLI Behavior

A parallel program consists of some number of processes, each involving some number of threads. Processes fall into two categories, depending on when they are created:

- **Initial process**

  A preexisting process from the normal run-time environment (that is, created outside the debugger) or one that was created as TotalView loaded the program.

- **Spawned process**

  A new process created by a process executing under the CLI's control.

TotalView assigns an integer value to each individual process and thread under its control. This *process/thread identifier* can be the system identifier associated with the process or thread. However, it can be an arbitrary value created by the CLI. Process numbers are unique over the lifetime of a debugging session; in contrast, thread numbers are only unique over the lifetime of a process.

Process/thread notation lets you identify the component that a command targets. For example, if your program has two processes, and each has two threads, four threads exist:

Thread 1 of process 1
Thread 2 of process 1
Thread 1 of process 2
Thread 2 of process 2

You would identify the four threads as follows:

1.1—Thread 1 of process 1
1.2—Thread 2 of process 1
2.1—Thread 1 of process 2
2.2—Thread 2 of process 2

Kinds of IDs

Multithreaded, multiprocess, distributed program contain a variety of IDs. Here is some background on the kinds used in the CLI and TotalView:

System PID This is the process ID and is generally called the PID. This ID usually has a value between 100 and 32,000. However, it can be higher on some systems.

Debugger PID This is an ID created by TotalView that lets it identify processes. It is a sequentially numbered value beginning at 1 that is incremented for each new process. Note that if the target process is killed and restarted (that is, you use the **dkill** and **drun** commands), the debugger PID does not change. The system PID, however, changes since the operating system has created a new target process.

System TID This is the ID of the system kernel or user thread. On some systems (for example, AIX), the TIDs have no obvious meaning. On other systems, they start at 1 and are incremented by 1 for each thread.

TotalView thread ID This is usually identical to the system TID. On some systems (such as AIX where the threads have no obvious meaning), TotalView uses its own IDs.

pthread ID This is the ID assigned by the Posix pthreads package. On most systems, this differs from the system TID. In these cases, it is a pointer value that points to the pthread ID.

Controlling Program Execution

Knowing what is going on and where you program is executing is straightforward in a serial debugging environment. Your program is either stopped or running. When it is running, an event such as arriving at a breakpoint can occur. This event tells the debugger to stop the program. Sometime later, you will tell the serial program to continue executing. Multiprocess and multithreaded programs are much complicated. Each thread and each process has its own execution state. When a thread (or set of threads) triggers a breakpoint, TotalView must decide what it should do about the other threads and processes. Some may stop; some may continue to run.

Advancing Program Execution

Debugging begins by entering a **dload** or **dattach** command. If you use the **dload** command, you must use the **drun** command to start the program executing. These three commands work at process level and cannot be used to start an individual threads. (This is also true for the **dkill** command.)

To advance program execution, you enter a command that causes one or more threads to execute instructions. The commands are applied to a P/T set. Because the set does not have to include all processes and threads, you can cause some processes to be executed while holding others back. You can also advance program execution by increments, *stepping* the program forward, and you can define the size of the increment.

Typically, debugging a program means that you have the program run, and then you stop it and examine its state. In this sense, a debugger can be thought of as tool that allows you to alter a program's state in a controlled way. And, debugging is the process of stopping the process to examine its state. However, the term "stop" has a slightly different meaning in a multiprocess, multithreaded program; in these programs, *stopping* means that the CLI holds one or more threads at a location until you enter a command that tells them to start executing again.

Action Points

Action points tell the CLI that it should stop a program's execution. You can specify four different kinds of action points:

- A *breakpoint* (see "dbreak" on page 142) stops the process when the program reaches a location in the source code.
- A *watchpoint* (see "dwatch" on page 225) stops the process when the value of a variable is changed.
- A *barrier point* (see "dbarrier" on page 137), as its name suggests, effectively prevents processes from proceeding beyond a point until other processes arrive. This gives you a method for synchronizing the activities of processes. (Note that barriers can only be applied to entire processes, not to individual threads.)
- An *evaluation point* (see "dbreak" on page 142) lets you programmatically evaluate the state of the process or variable when execution reaches a location in the source code. Evaluation points typically do not stop the process; instead, they perform an action.

NOTE Extensive information on action points can be found in the TotalView User's Guide.

Each action point is associated with an *action point identifier*. You use these identifiers when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1. The second ID is 2, and so on. These numbers are never reused during a debugging session.

The CLI and TotalView only let you assign one action point to a source code line. However, neither limits the complexity of an action point.

Chapter 5

Type Transformations

In some cases, TotalView cannot display data in the way you intended. For example, compilers do not place information within an executable that defines the way data is used for C++ STL container types. Instead, they describe the implementation of the STL containers, which is normally not of interest to anyone using these classes.

The type transformation system allows you to describe how TotalView should display these kinds of data types.

Type Transformation Defined

TotalView, as it is shipped, knows how to display a variety of compiler-defined data types and, depending on your programming language, **struct**s, or user-defined types. If your data embeds pointers, diving on a pointer displays the "pointed to" data. If you are using more complicated data types or are using the STL data types, you will need to create a *prototype* that tells TotalView how it should display this data. This is because TotalView displays your data as the compiler understands it to be defined.

The prototype you can define names a set of Tcl callback functions that TotalView invokes when it displays your data. These routines tell TotalView that is should display the information in the way you say, which will either be a **struct** or an array. This process of telling TotalView how to display your data is called *type transformation*.

For instance, the next figure shows how TotalView displays an object of **class vector<int>** if you used the C++ **std::vector** used by g++.

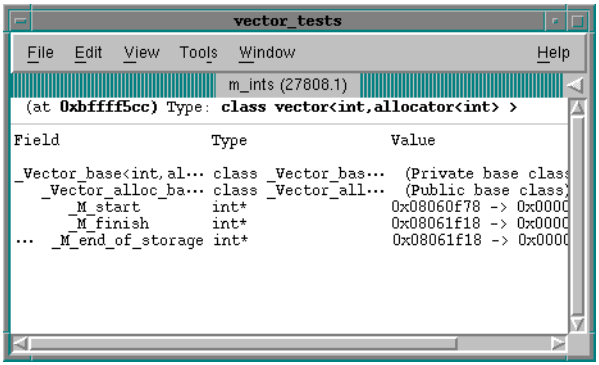

FIGURE 26: **Unmapped std::vector<int>**

Although this is exactly what the compiler told TotalView, most users would like this information displayed as an array. After instantiating a prototype for this type, TotalView can display this information as follows:

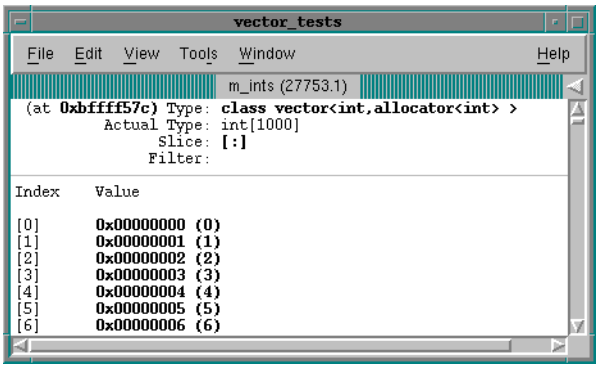

FIGURE 27: **Mapped std::vector<int>**

Notice that TotalView shows the vector's current size and allows the user to slice and filter the data. The user could even send this data to the Visualizer.

Creating Type Transformations

Creating and using a type transformation involves the following three steps:

1. Define a set of Tcl callback functions. As you will see, the kind of data being mapped determines the number and kind of callback functions that you will use.
2. Use the **TV::prototype** command to create a new prototype object. The callback functions are named as properties of this object.

   You will define a separate prototype for each data type being mapped.
3. Individually add your prototype to each image object in your process.

When TotalView parses the debug information created by your compiler, it checks the type name. If the type name matches the prototype's name (which is actually regular expression), TotalView uses your callbacks to redefine the type.

Later, when TotalView displays this data, it uses the callbacks to extract information from the specific instance. For example, the run-time bounds for an array depend on the array being displayed.

Using Type Transformation

Creating a type transformation requires a sophisticated understanding of the way the compiler stores information. However, using the type transformation is a simple process of instantiating the callback functions. At many sites, the person who creates prototypes is different than the person who actually uses them. That is, a prototype builder will create libraries of prototypes that others will use.

For example, after someone writes the type transformation functions for a **std::vector** type and places them in a **.tcl** file, anyone can use Tcl's **source** command to install the type transformation. In most cases, however, the transformation is placed in an initialization file. The type mapping macros end with two CLI statements that instantiate the type transformation.

Here, for example, are the statements to install the **std::vector** type transformation:

```
set proto_id [TV::prototype create array]

TV::prototype set $proto_id \
    name              {^(class|struct) (std::)?vector *<.*>$}\
    language          C++                                   \
    validate_callback vector_validate                       \
    type_callback     vector_type                           \
    address_callback  vector_address                        \
    typedef_callback  vector_typedef                        \
    rank_callback     vector_rank                           \
    bounds_callback   vector_bounds
```

This example invokes the **TV::prototype** command twice. The first time obtains a TotalView-generated identifier for the prototype. (This is similar to a handle.) The second associated properties with a prototype ID.

Once the type transformation code is loaded, TotalView automatically applies it and you can forget that it exists. The only exception is when you want to see the underlying implementation type. You can do this by using the **<internal>** string to cast the type in a Variable Window. For example, the following figure shows how the **std::vector** appears when it is cast to class vector**<int,allocator<int>> <internal>**.

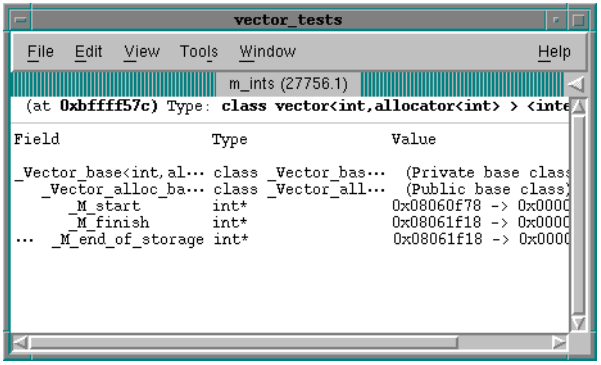

FIGURE 28: **Internal View of std::vector<int>**

Defining Prototypes

A prototype contains the names of Tcl callback functions that are used at various times in the type transformation process, as follows:

- When TotalView is creating a type object, it validates the prototype by using a callback function that checks if the name of the prototype matches that of the data type.
- When TotalView is about to display an object with this prototype, it invokes callbacks that extract object properties.
- If a **typedef** is being set up, TotalView calls the **typedef_callback**.

As you will see, there are two styles of prototypes: arrays and **struct**s. As part of the process of defining the prototype, you must tell TotalView which style you are creating.

The following properties are (or can be) included in all prototypes. Other sections in this chapter present the routines that are associated with these callbacks.

name (required) The regular expression that TotalView uses to match types. It must be anchored (that is, start with a " ^ " character.

language (required) The language for this prototype. It indicates how TotalView parses the program's data and generates bounds and indices for it.

validate_callback (required) TotalView invokes this callback whenever a type that matches the prototype's name is defined. It returns a Boolean value indicating if it should be applied. This callback lets you have more than one prototype match a type name, and then investigate the type to determine if TotalView should apply the prototype.

The call structure for a validation callback is:

validate_callback *type_id*

where *type_id* is the type identifier for the type being prototyped.

address_callback Generates the address of the object's elements at run time. It returns either an absolute address, or an addressing expression that is appended to the address of the object to give the address of the field. (Returning an expression is the preferred method.)

For example, you might use a callback if the original data structure contains information on where the next data instance resides.

If you are creating a type for a distributed array, this procedure returns a two-element list. For more information, see "The Distributed Addressing Callback" on page 113.

The call structure for an address callback is:

address_callback *type_id object_addr index* [*replication*]

where:

type_id: is the type identifier for the type being prototyped.

object_addr: is the address of the object.

index: is the index string for the array element or the index of the field in the structure.

replication: is only used for distributed array objects. It indicates that the result is an address and an index into the distribution to determine the process within which this element resides, since this is a distributed array.

type_callback (required) TotalView invokes this callback when the prototype is modifying a type. Its format is:

type_callback *type_id*

A*rray prototypes*: Returns a value that is the type identifier for one of the array's elements.

S*truct prototypes*: Returns a list in the format of the **struct_fields** property that describes the **struct** type's fields. If a field in the type requires a callback, the addressing section of its field description should be the string **callback** rather than an addressing expres-

sion. In this case, TotalView uses the **address_callback** to generate the address of this field.

typedef_callback Defines an *new_type_id* in terms of the *old_type_id*. You would use this callback when the prototype modifies *old_type_id*.

typedef_callback *new_type_id old_type_id*

TotalView ignores any returned value.

The following three callbacks are only used with array prototypes:

bounds_callback Returns either a string that specifies the bounds statically or a callback that TotalView calls when the object's address is known. If you are naming a callback, its call structure is:

bounds_callback *type_id object_address*

The string returned by the callback describes the current bounds. This string must be in program's programming language. For example:

C: [2][40]
Fortran: (-2:10,-5:5)

If the bounds start with a bracket "[" or a parenthesis "(", TotalView assumes the bounds are static; otherwise, TotalView assumes that the returned value is the name of a callback function.

As the bracket characters ([ ]) are special characters in Tcl, you must escape them even in strings; for example \[20\] rather than [20].

rank_callback Returns the array's rank. TotalView uses this callback when the prototype modifies a type. Its call structure is:

rank_callback *type_id*

distribution_callback
Returns a list of process or thread identifiers that represent (in order) the processes/threads in which elements of this array exist. Only use this callback when your array is distributed over multiple processes. Its call structure is:

distribution_callback *type_id object_address*

TotalView calls this callback with a replication index. The returned value must have an index into the distribution as well as an address.

Before using the **TV::image add prototype** command to add a prototype to an image, you must use the **TV::prototype set** command to set its properties. After a prototype is added to an image, you can no longer change its properties.

Objects Used in Type Transformation

A type transformation uses the following CLI commands. You will find extensive information about these commands in Chapter 6.

- **TV::process**, which accesses a process.

  A process represents a single UNIX process. The process can have many threads of control within it. A process always references at least one image.

- **TV::image**, which accesses an image.

  An image is the object that describes a single executable file—either the executable image or a shared library. An image owns a set of types and a set of prototypes.

  You can ask a process which set of images it is currently using. Note that this set can change if the process calls **dlopen()** to load a new shared library, or **dlclose()** to remove one.

  NOTE You can determine which prototypes are associated with an image by using the "TV::image get *image_id* prototypes" command. This command returns a list of the prototype IDs added to the image.

  Within an image, you can look up types by name. A single named type may have many internal types associated with it because compilers often output a type definition in the debug information for each source file. This means that *lookup* operation returns a list of all of the types with the requested name. For example:

  ```
  d1.<> TV::image lookup 1|1 types <integer>
  1|6
  ```

  An image owns a set of types and refers to a set of prototypes.

- **TV::type**, which acceses a type.

  A type object holds the information about a single type. The type belongs to an image object and cannot be used outside that image.

Creating a struct Type Transformation

This section describes a type transformation that is used with the g++ **std::list** class. Here is how this class is defined:

```
template <class _Tp>
struct _List_node {
    typedef void*       _Void_pointer;
    _Void_pointer       _M_next;
    _Void_pointer       _M_prev;
    _Tp                 _M_data;
};
```

This template definition does not contain information that TotalView can use to determine that object pointed to by the *next* and *prev* pointers are **_List_node** objects. Creating a transformation will allow TotalView to display this information in a more orderly way.

Validating the Type: the list_validate Procedure

The first operation that TotalView will perform is to look at the datatypes used in your application and decide which of them will be mapped. The validation callback routine checks insures the type matches this code's expectations. This routine will need to return the structure's definition.

The **list_validate** routine casts the types in one of the subtypes of **_List_node<foo>** that contains **void *** pointers. This allows them to be interpreted as **_List_node<foo> *** pointers.

NOTE This code reflects the "libstd" c++ implementation of "list<>" used by Linux g++ compilers.

```
proc list_validate {instance_id} {
    # Check for _M_next, _M_prev, and _M_data.
    set fields [TV::type get $instance_id struct_fields]
    set matched_fields 0
```

```
    foreach field $fields {
        set field_name [lindex $field 0]

        switch -- $field_name {
            _M_next -
            _M_prev -
            _M_data {
                incr matched_fields
            }
        }
    }

    return [expr $matched_fields==3]
}
```

Redefining the Type: list_type Procedure

The **list_type** function returns the structure definition that TotalView will use. This definition essentially duplicates the type's definition except that two of the type IDs become pointers to the real target type rather than pointers to void.

```
proc list_type {instance_id} {
    set instance_name [TV::type get $instance_id name]
    set target_name "$instance_name *"
    set ptr_id [TV::image lookup \
      [TV::type get $instance_id image_id] types $target_name]

    # In case there is more than one type that matches,
    # choose the first.
    set ptr_id [lindex $ptr_id 0]

    # Walk over the fields changing the types of appropriate
    # ones. Note that this does not touch the addressing. It
    # merely changes the types of some of the fields.
    set original_fields [TV::type get $instance_id struct_fields]

    foreach field $original_fields {
        set field_name [lindex $field 0]

        if {$field_name == "_M_next" || \
                    $field_name == "_M_prev"} {
```

```
            set field [lreplace $field 1 1 $ptr_id]
        }
        lappend result_fields $field
    }

    return $result_fields
}
```

Creating the Prototype

The following two commands create a prototype object and set the object's properties. The argument to the **TV::prototype create** command indicates if you are defining an array or **struct** prototype. The **TV::prototype set** command initializes the object's properties.

```
set proto_id [TV::prototype create Struct]

TV::prototype set $proto_id \
    name                {^(class|struct) _List_node *<.*>$}    \
    language            C++                                    \
    validate_callback   list_validate                          \
    type_callback       list_type
```

Making a Callback for a Structure Element

You could also request that an addressing callback be made for a structure element. If you do, it is called when TotalView displays a specific instance of the type. You can do this if you:

- Specify the addressing of the field as **callback** in the type callback.
- Provide an **address_callback** on the structure prototype.

The address callback will be called with the following arguments:

- The *type_id* for the type.
- The *address* of the instance.
- The *index* of the field whose address is required.

The **address** callback should return either an address expression or an absolute address.

The following example does not create an addressing callback because the structure field's address does not depend on the specific instance of the

structure. This means TotalView will have no problem evaluating it with own addressing expression.

Applying Prototypes to Images

The properties of a prototype do not affect an object until you add it to an image by using the **TV::image add** command. Here is this command's format:

TV::image add *image_id* **prototype** *proto_id*

When you load a prototype, TotalView looks within its **TV::image_load_callbacks** variable. This variable contains a Tcl list of procedure names, each of which is invoked by TotalView whenever a new image is loaded by TotalView. This could occur when:

- A user invokes a command such as **dload**.
- TotalView resolves dynamic library dependencies.
- User code uses **dlopen()** to load a new image.

TotalView always initially sets this variable to a list containing the name of the **TV::propagate_prototypes** routine. This function applies all prototypes to every new image as it is loaded. Therefore, if you create prototypes before any image is loaded (for instance, by executing prototype creation code in a **.tvdrc** file), **TV::propagate_prototypes** tells TotalView to apply all prototypes to all images as they are loaded.

By adding other function names to the **TV::image_load_callbacks** list, you can tell TotalView to take additional actions. For instance, you could define prototypes for g++ STL only when **/lib/libstdc++.so** is loaded.

TotalView invokes the functions in order, beginning at the first function in the list.

If you create prototypes after images are loaded, and you want the prototypes to be applied to the existing images, you will need to apply them explicitly to these images. The following example shows the kind of

procedure that you could write to apply prototypes to all images in the current focus.

```
# Given the identifier for a prototype object, add it to all
# images used by processes in the current focus.
proc apply_prototype_to_focus {proto_id} {
        set processes [TV::focus_processes]

    # Find the set of images.
    foreach process $processes {
        set images [TV::process get $process image_ids]
        foreach image $images {
            set image_names($image) true
        }
    }

    # Now add the prototype to the image.
    foreach image [array names image_names] {
        TV::image add $image prototype $proto_id
    }
}
```

NOTE You can change any of a prototype's properties before its added. As soon as you add one, however, its properties are frozen and TotalView does not allow you to alter them.

Here is how you would invoke this function:

```
apply_prototype_to_focus $proto_id
```

Initializing TotalView After Loading an Image

Immediately after TotalView loads an image and just before it runs it, TotalView executes the routines listed in its **TV::process_load_callbacks** lists. TotalView invokes these callbacks after it invokes the routines in the **TV::image_load_callbacks** list. By default, the value of the **TV::process_load_callbacks** variable is **{TV::source_process_startup}**. The routines in this second callback list are only called once even though your executable may use many images (one for each shared library).

TotalView initializes the **TV::image_load_callbacks** list variable with the name of the **TV::source_process_startup** routine. This routine looks for a

file with the same name as the newly loaded process's executable image that has a **.tvd** suffix appended to it. If the file exists, TotalView executes the commands contained within it.

This function is passed a single argument that is the ID for the newly created process.

An Array-Like Example

This section contains an extended example that performs a type transformation for the **std::vector** type. This example has two parts. The first part uses CLI commands that install the callback functions. The second contains several utility functions that simplify the logic of the callback functions.

NOTE As an aid in understanding the function listings, these utility functions are displayed in bold type.

This type transformation uses a global array to store information about the vector type's being remapped. This array, which is indexed by the type identifier, contains the following three-element list:

- The type identifier for the type of the elements of the array.
- The offset of the **_M_start** field.
- The offset of the **_M_finish** field.

Indicating if a Type Is Mapped: The vector_validate Callback

The **vector_validate** procedure checks that this named type matches the expected specification. This function expects the following input:

```
_Vector_base<int,allo.. class _Vector_base<..
            (Private base class)
    _Vector_alloc_base.. class _Vector_alloc..
            (Public base class)
        _M_start int*
        _M_finish int*
        _M_end_of_storage int*
```

It also extracts other information about the type and saves it for later use.

```
proc vector_validate {instance_id} {
    global _vector_type_info

    set base_id [ultimate_base $instance_id]
    set fields [TV::type get $base_id struct_fields]

    # Prepare the information that will be saved.
    set typeinfo [list {} {} {}]

    # Search for _M_start and _M_finish.
    foreach field $fields {
        set name [lindex $field 0]
        set addressing [lindex $field 2]

        switch -- $name {
            _M_start{
                set typeinfo [lreplace $typeinfo 1 1 \
                        [extract_offset $addressing]]
                # Also extract the type
                set field_typeid [lindex $field 1]
                set typeinfo [lreplace $typeinfo 0 0 \
                        [TV::type get $field_typeid target]]
            }
            _M_finish {
                set typeinfo [lreplace $typeinfo 2 2 \
                        [extract_offset $addressing]]
            }
        }
    }
    # Check that the target type was found.
    if {[lindex $typeinfo 0] == ""} {
        return false
    }

    set _vector_type_info($instance_id) $typeinfo
    return true
}
```

This procedure begins by finding the ultimate base class of the vector type, and then checks that the data type has the named fields. It also checks that the data type really is the class being transformed. Just before returning, it places information needed by other procedures into the **_vector_type_info** global array.

The **vector_validate** procedure returns false if it cannot match the target type. This tells TotalView that it should not apply this prototype to the type. TotalView will, however, continue searching for other prototypes whose name matches the type's name.

NOTE This procedure could fail if some code is compiled for g++, and some by a vendor's compiler.

Returning the Type: The vector_type Callback

The **vector_type** procedure returns the type ID for the target type. This is a trivial process because the type ID was previously set by the **vector_validate** procedure.

```
proc vector_type {instance_id} {
    global _vector_type_info
    return [lindex $_vector_type_info($instance_id) 0]
}
```

Returning the Rank: The vector_rank Callback

The **vector_rank** procedure is trivial because a vector is a one-dimensional array. Consequently, the returned rank (that is, the number of dimensions) is always 1.

```
proc vector_rank {type_name} {
    return 1
}
```

Returning the Bounds: The vector_bounds Callback

The **vector_bounds** procedure returns the bounds of the object at an address. This returned value is a string in the syntax of the language in which the type was defined. TotalView calls this procedure whenever it is about to display a **std::vector** object. This example must use a callback since the vector type has dynamic bounds. If the type being mapped had static bounds, the routine could just return a string.

The **vector_bounds** procedure calculates the bounds by:

- Reading the base and end pointers.
- Subtracting one of these values from the other.
- Dividing this value by the element size.

Once again, this process is pretty simple because the **vector_validate** routine wrote much of this information into the **_vector_type_info** global variable.

```
proc vector_bounds {type_id address} {
    global _vector_type_info

    set vti $_vector_type_info($type_id)

    set start [read_store [expr $address+[lindex $vti 1]]]
    set finish [read_store [expr $address+[lindex $vti 2]]]

    # Extract the length of the target type.
    set type_size [TV::type get [lindex $vti 0] length]

    set delta [expr ($finish-$start)/$type_size]

    return "\[$delta\]"
}
```

Returning the Address: The vector_address Callback

The **vector_address** procedure computes the address of an array element. TotalView calls it for each array element being displayed.

TotalView supplies the following three parameters when it calls your procedure:

- The type ID.
- The array's starting address.
- The index for which the address is required.

The callback returns an addressing expression that allows TotalView to generate the address of a vector element. In this example, the **vector_address** function returns an addressing expression. Here is an example:

```
{{addc 4} indirect {addc 4}}
```

NOTE See "Addressing Expressions" on page 114 for information on the operators and opcodes that you can use.

While an address procedure could return an absolute address, it should instead return an addressing expression. Returning a procedure is better

because TotalView can then recalculate the address when you dive through an element in a Variable Window. If you had returned an absolute address, TotalView would use the address in the new Variable Window, even if the vector has changed.

```
proc vector_address {type_id address indices} {
    global _vector_type_info

    set vti $_vector_type_info($type_id)

    set type_size [TV::type get [lindex $vti 0] length]
    set offset [expr $type_size*$indices]

    # We don't bother to worry about adding zero because
    # TotalView will remove any unnecessary additions.
    set result "{{addc [lindex $vti 1]} indirect {addc $offset}}"

    return $result
}
```

The typedef Callback

If your program defines a new type using a **typedef** for a mapped data type, TotalView calls the **typedef** callback. This callback lets you copy state information that is indexed by the type identifier since the new type will have a different type identifier from the old one.

The following example copies vector information associated with one ID to another"

```
proc vector_typedef {new_id old_id} {
    global _vector_type_info
    set _vector_type_info($new_id) $_vector_type_info($old_id)
}
```

Utility Procedures

This section presents three utility procedures that simplify the type mapping process for this type:

- The read_store Utility Procedure
- The ultimate_base Utility Procedure
- The extract_offset Utility Procedure

The read_store Utility Procedure

The **read_store** procedure parses the output of the CLI's **dprint** command, and then stores and returns a value. This procedure reads a value from an absolute address.

```
proc read_store {address {type void}} {
    set res [capture dprint "*($type *)$address"]

    # Strip out just the value
    regexp {^.*= ([^ ]*)} $res {} res

    return $res
}
```

The regular expression uses the {} empty string to indicate that the CLI should ignore the entire string that matches. The portion of the regular expression within the parentheses (that is, the value) is assigned to **res**, which is the function's return value.

The ultimate_base Utility Procedure

The **ultimate_base** utility procedure finds the ultimate base class of a class in a single inheritance chain by extracting the structure fields and iterating over the information until it arrives at a class that does not have a base class.

```
proc ultimate_base {type_id} {
    while {1} {
        set fields [TV::type get $type_id struct_fields]
        set first_member [lindex $fields 0]
        set properties [lindex $first_member 3]

        if {[regexp {base class} $properties] == 0} {
            return $type_id
        } else {
            set type_id [lindex $first_member 1]
        }
    }
}
```

The extract_offset Utility Procedure

The **extract_offset** routine evaluates and addressing expression so that it can return an offset for a data item. This example assumes that the addressing expression uses **addc**, and returns the argument of the **addc** opcode. This argument is the offset being added.

```
proc extract_offset {addressing_expr} {
    if {[llength $addressing_expr] != 1} {
        return 0
    }

    # Unwind the list.
    set addressing_expr [lindex $addressing_expr 0]
    if {[lindex $addressing_expr 0] != "addc"} {
        return 0
    } else {
        # return addc's operand
        return [lindex $addressing_expr 1]
    }
}
```

Creating the Prototype

This section describes the two calls that must be made to the **TV::prototype** command. The first tells TotalView to create a prototype and the second defines this prototype's properties.

The **TV::prototype create** command creates a new prototype object. Its sole argument indicates if you are defining an array or **struct** prototype. This ID for this object acts as a handle that you use when working with the prototype. The second command, **TV::prototype set**, defines the prototype's properties.

Here are the commands that set up the prototype for **std::vector**:

```
set proto_id [TV::prototype create array]

TV::prototype set $proto_id \
    name                {^(class|struct) (std::)?vector *<.*>$}\
    language            C++ \
    validate_callback   vector_validate \
    typedef_callback    vector_typedef \
```

```
type_callback       vector_type \
rank_callback       vector_rank \
bounds_callback     vector_bounds \
address_callback    vector_address
```

The first statement creates the prototype and assigns the returned ID to the **proto_id** variable. The next statement sets its properties. While these statements create the new prototype and define its properties, the prototype is simply a definition of what can be done. This changes when you activate it by adding it to an image. For example, you could use the **apply_prototype_to_focus** routine described in "Applying Prototypes to Images" on page 98 to activate it. Here is how you would use this function:

```
apply_prototype_to_focus $proto_id
```

Distributed Arrays

Many parallel applications operate on arrays that are distributed across the parallel program's processes. In these applications, each process operates on a local component of a larger array. While each process has parts of the data, you sometimes need to integrate all this data into a common Variable Window to examine the data as it is being manipulated in all processes.

If the data is a standard C or Fortran array, you can use the Variable Window's **View > Laminate** command. However, if the data's type is more complicated, you will need to use the type mapping's **distributed_callback** property. This property lets you see the distributed array in its own global index space.

This section uses the **mandel.c** program that is listed in Appendix C, "*Distributed Array Type Mapping*". This is a simple MPI program that distributes an array cyclically in one dimension over all processors. Each processor calculates one part of the Mandelbrot set. Of course, **mandel.c** is not the best way to compute the Mandelbrot set, but the point here is not the Mandelbrot set, but the distributed array.

This program distributes an array of structures cyclically across all of the processes in the MPI job in the second dimension. On each processor,

however, individual columns are held contiguously. The Mandlebrot program's **local_column()** and **owner_of()** procedures provide the transformation from a global array index to the local index and node. Figure 29 shows how TotalView displays the structure's array data on one node.

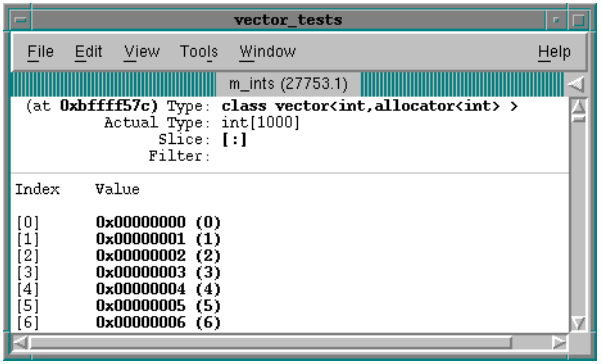

FIGURE 29: **Standard Display of struct "a" as an Array**

However, type transformation lets TotalView display the entire array in its global index space. Figure 30 shows is a Variable Window that shows this reassembled data.

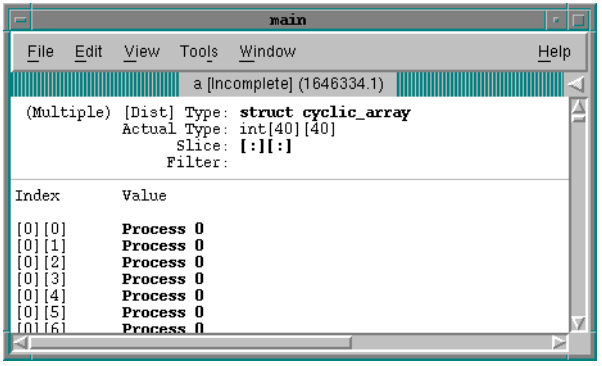

FIGURE 30: **Reassembled Display**

Since TotalView knows that the array is distributed, you can even show individual node information. For example, tFigure 31 slices out the first element of each column, and displays which node owns it.

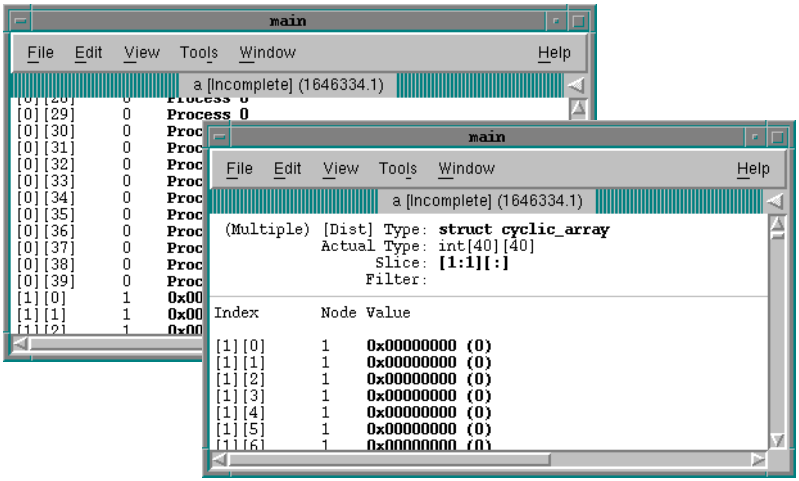

FIGURE 31: **A Slice of the Reassembled Array**

Visualizing a Distributed Array with Node Information

The reassembled array can also be sent to the Visualizer. Figure 32 shows that the reassembled data does generate a Mandelbrot set.

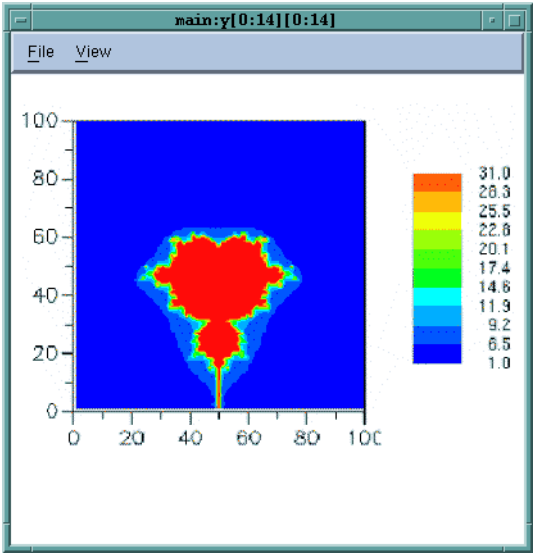

FIGURE 32: **Visualization of the Reassembled Array**

The Type Transformation for mandel.c

The commands that install a type transformation for the distributed array within **mandel.c** are:

```
set proto_id [TV::prototype create Array]

TV::prototype set  $proto_id                              \
    name                  {^struct cyclic_array$}         \
    language              c                               \
    validate_callback     da_validate                     \
    typedef_callback      da_typedef                      \
    type_callback         da_type                         \
    rank_callback         da_rank                         \
    bounds_callback       da_bounds                       \
    address_callback      da_address                      \
    distribution_callback da_distribution

apply_prototype_to_focus $proto_id
```

This chapter will only present **validate_callback**, **address_callback**, and the **distribution_callback** procedures. The other callbacks and the related utility functions are either identical or similar to those that have already been discussed. (A complete listing of these callbacks is located in "The cyclic_array.tcl Type Mapping File" on page 286.) The most important differences between this distributed example and the other array example are:

- The prototype also has a **distribution_callback** property. This callback provides TotalView with the set of processes over which the array is distributed.
- If an array prototype has a distribution callback, you must:
  - Add a replication argument to the addressing callback.
  - Provide an index into the distribution.
  - Create an addressing operation that indicates the process or thread from which TotalView will fetch the data and its address.

Validating the Data Type: The da_validate Function

The **da_validate** function checks the data's type is the one to be mapped and that it contains the required fields. These fields are:

```
#   struct cyclic_array
#   {
#       int * local_elements;
#       int local_count;
#       int global_count;
#       int numprocs;
#       int myproc;
#   };
```

Here is the validation procedure:

```
proc da_validate {instance_id} {
    global _da_info

    set fields [TV::type get $instance_id struct_fields]
    # We'll save four properties of each type:
    #   The offset of the pointer to the local array.
    #   The target type identifier.
    #   The target type size.
    #   The offset of the local_count.
    #   The offset of the global count.
    set typeinfo [list {} {} {} {} {}]
    set matched 0

    foreach field $fields {
        set name [lindex $field 0]
        set addressing [lindex $field 2]

        switch -- $name {
            local_elements  {
                set typeinfo [lreplace $typeinfo 0 0 \
                        [extract_offset $addressing]]
                # Extract the target type too.
                set field_typeid \
                        [TV::type get [lindex $field 1] target]

                set typeinfo [lreplace $typeinfo 1 1 $field_typeid]
                set typeinfo [lreplace $typeinfo 2 2 \
                        [TV::type get $field_typeid length]]

                incr matched
            }
```

```
                local_count {
                    set typeinfo [lreplace $typeinfo 3 3 \
                            [extract_offset $addressing]]
                    incr matched
                }

                global_count {
                    set typeinfo [lreplace $typeinfo 4 4 \
                            [extract_offset $addressing]]
                    incr matched
                }
            }
        }

        if {$matched != 3} {
            return false
        }

        set _da_info($instance_id) $typeinfo
        return true;
    }
```

The da_distribution Callback Procedure

The following procedure returns a list of process and thread identifiers over which this array is distributed. In this simple example, any of these arrays are distributed over all processes.

```
proc da_distribution {type_id address} {
    #
    # For the moment we assume this is all items in our
    # workers group.
    global GROUP WGROUP _da_nprocs

    # Choose the first process in the focus set.
    set proc [lindex [TV::focus_processes] 0]

    # Find the relevant worker group identifier.
    set group_id $WGROUP($proc)

    # Extract the member identifiers from the worker
    # contents.
    set res [lrange $GROUP($group_id) 1 end]
```

```
    # Save the number of processes for later.
    set _da_nprocs [llength $res]

    return $res
}
```

The **da_distribution** procedure defines the set of processes or threads that contribute data to the distributed object. It returns a list containing the names of these objects. The order of the processes and threads in this list is important since the value returned by the address callback (described in the next section) to describe the process or thread is an index into this list.

The Distributed Addressing Callback

The **da_address** procedure computes the address of an array element.

```
proc da_address {type_id address indices replication} {
    # Each element lives in only one place, so we return a null
    # result if asked for other places for it.
    if {$replication != 0} {
        return ""
    }

    global _da_info _da_nprocs

    set typeinfo $_da_info($type_id)
    set bound [read_store \
                    [expr $address + [lindex $typeinfo 4]] int]

    set distributed_index [lindex $indices 0]
    set other_index [lindex $indices 1]
    set node [expr $distributed_index%$_da_nprocs]
    set local_index [expr $distributed_index/$_da_nprocs]

    set element_size [lindex $typeinfo 2]

    #
    # We have to work out the whole address.
    #
    set delta [expr $element_size* \
                ($other_index+$bound*$local_index)]
```

```
    return \
        "$node {addc [lindex $typeinfo 0]; indirect; addc $delta}"
}
```

The **da_address** routine begins by checking if TotalView is requesting extra places where this data object might live—this occurs when your program is updating a value in a distributed array. Since some elements of the array could be replicated, TotalView attempts to update them everywhere they may reside. In this example, however, the array data is not replicated. Consequently, the routine returns a null string for all replication values except the first.

For the first (and only) replicated element, **da_address** performs the same calculations as the Fortran **local_address()** and **iwho()** procedures. These routines:

- Compute the index into the local array and the node index for an element.
- Compute the element's full offset, including the address in the undistributed dimension.
- Return the node index and an addressing expression that allows TotalView to locate the data object.

The procedure uses the **gSizeof_Element**, **gDa_Nprocs**, and **gArray_Bound** global variables set up by earlier callbacks.

Addressing Expressions

An addressing expression tells TotalView how to address a variable, a field in a structure, or an element in an array. Using the **TV::type get** *image_id* **struct_fields** command, you can obtain the addressing expressions being used by TotalView. You will also need to pass an addressing expression back to TotalView that uniquely describes how TotalView will locate data elements.

Wherever possible, a callback should generate an addressing expression rather than returning an absolute address since the use of addressing expressions is much more efficient.

TotalView uses addressing expressions as results from the TCL callback functions when handling variables with prototyped types; they are generated by TotalView to describe the addressing required to reach a field in a composite type. They can then be seen as part of the result of the **TV::type get struct_fields** command.

An addressing expression is a set of operations for a stack-machine that evaluate an address. These operations are appended to those that TotalView has already used to reach the specific instance of the object with that type.

All addressing expressions should be wrapped in **{}** and can be structured as arbitrary lists. When generating addressing expressions, TotalView formats each opcode/operand pair as one sublist containing the expression; for example:

```
d1. <> TV::type get 1|11 struct_fields
{bit_enum 1|12 {{bitfield_index {2>>0 unsigned}} } {}}
{wide_enum 1|13 {{bitfield_index {30>>2 unsigned}} } {}}
```

TotalView ignores the list structure when it reads an addressing expression generated by user code.

A simple numeric operand is described below as *opd*, which is a single decimal or hexadecimal (0x...) number.

The **bitfield_index** and **bitfield_value** opcodes are more complicated and are encoded as:

size>>*shift* [*un*]*signed*

where:

size is the size in bits of the field and *shift* is the shift required to justify the field at the low-significance end of the word. This field is sign-extended if tagged as signed; otherwise, it remains unsigned.

The following tables use the following notation and abbreviations:

TOS	Top of Stack.
memory[*n*]	The world value read from the thread address space at address *n*.

stack[*n***]**	The value of the *n*th element of the stack, where **stack[0]** is the top of the stack.

The following opcodes do not use values on the stack. However, they do push the stack.

TABLE 4: **Operations with Nonstack Opcodes**

Opcode	Meaning
ldac *opd*	Load the address of the constant *opd*
ldal *opd*	Load the address of the local variable whose offset from the frame pointer is *opd*
ldar *opd*	Load the address of register *opd*
ldatls *opd*	Load the address of the thread local storage object at offset *opd* in the thread local space
ldc *opd*	Load the constant *opd*
ldgtls *opd*	Load the address of the general thread local storage object whose key is *opd*
ldl *opd*	Load the value of the local variable whose offset from the frame pointer is *opd*
ldm *opd*	Load the value stored in memory at address *opd*
ldr *opd*	Load the contents of register *opd*

The following table lists opcodes with operands that also use data from the TOS.

TABLE 5: **Opcodes with Operands that Use the TOS (Top of Stack)**

Opcode	Meaning
addc *opd*	TOS = TOS + *opd*
bitfield_index *bitopd*	Load the address of the bit field whose store address is in the TOS. This must be the last opcode in an addressing expression.
indirect_small *opd*	Load *opd* bytes from **memory[TOS]** and zero extend.
ldnl *opd*	Load the value at address **TOS+opd**.

For opcodes without operands, all data comes from the stack.

TABLE 6: Operands Without Opcodes

Operand	Meaning
abs	TOS = abs (TOS)
and	TOS = TOS & stack[1]
div	TOS = TOS / stack[1]
drop	Pop TOS and discard
dup	Push TOS
indirect	TOS = memory[TOS]
minus	TOS = TOS - stack[1]
mod	TOS = TOS % stack[1]
mul	TOS = TOS * stack[1]
neg	TOS = - TOS
not	TOS = ~ TOS
or	TOS = TOS | stack[1]
over	Push the second entry on the stack
plus	TOS = TOS + stack[1]
rot	Rotate the top three stack entries.
shl	TOS = TOS << stack[1]
shr	TOS = TOS >> stack[1] (unsigned shift)
shra	TOS = TOS >> stack[1] (signed shift)
swap	Swap top two stack entries
xor	TOS = TOS ^ stack[1]

The following special opcode is most often used in addressing expressions that are appended to existing addressing expressions:

TABLE 7: Special Opcode

Opcode	Meaning
remove_indirection	Removes an indirection operation from the tail of the previous addressing expression; this is useful when you for backing up from data to a dope vector. (See the Glossary for more information.)

Debugging Tcl Callback Code

The first step in debugging a callback is to make sure that the function can works when it is not used in a type transformation callback; that is, if the function works when it is not used within a type mapping, it should work within one. Many procedures do not need to be tested within TotalView and can instead be tested directly by using a Tcl interpreter such as **tclsh**. In this environment, you can use standard Tcl debugging tools.

If one of your procedures throws a Tcl error inside a TotalView callback, TotalView prints a Tcl stack backtrace. For example:

```
CLI callback 'dfocus {1.1} {vector_address
    {class vector<int,allocator<int> >} 0xbffff9dc {0 } }' failed
    can't read "foo": no such variable while executing
    "set type_sizes $foo"
        (procedure "vector_address" line 6)
        invoked from within
    "vector_address {class vector<int,allocator<int> >}
    0xbffff9dc {0 } " invoked from within
    "dfocus {1.1} {vector_address {class
    vector<int,allocator<int> >}
    0xbffff9dc {0 } }"
```

If neither of these techniques helps, you will need to print values from your Tcl code as it executes so that you can see what is happening. You might want to use a debugging procedure such as:

```
proc my_puts {{string ""}} {
    global gMy_Debug

    if {$gMy_Debug} {
        puts $string
    }
}
```

The **my_puts** function will only print a message if the **gMy_Debug** global variable is set to **true**. This lets you enable and disable printing information by changing **gMy_Debug**'s value.

Chapter 6

CLI Commands

This chapter contains detailed descriptions of CLI commands.

Command Overview

This section lists all of the CLI commands. It also contains a short explanation of what each command does.

Environment Commands

The CLI commands in this group provide information on the general CLI operating environment:

- *alias*: Creates or views pseudonym for commands and arguments.
- *capture*: Allows commands that print information to instead send their output to a variable.
- *dlappend*: Appends list elements to a TotalView variable.
- *dset*: Changes or views values of TotalView state variables.
- *dunset*: Restores default settings of TotalView state variables.
- *help*: Displays help information.
- *stty*: Sets terminal properties.
- *unalias*: Removes a previously defined alias.

CLI Initialization and Termination

These commands initialize and terminate the CLI session, and add processes to CLI control:

- *dattach*: Brings one or more processes currently executing in the normal run-time environment (that is, outside TotalView) under TotalView control.
- *ddetach*: Detaches TotalView from a process.
- *dkill*: Kills existing user processes, leaving debugging information in place.
- *dload*: Loads debugging information about the program into TotalView and prepares it for execution.
- *drerun*: Restarts a process.
- *drun*: Starts or restarts the execution of user processes under control of the CLI.
- *exit, quit*: Exits from TotalView, ending the debugging session.

Program Information

The following commands provide information about a program's current execution location and allow you to browse the program's source files:

- *ddown*: Navigates through the call stack by manipulating the current frame.
- *dlist*: Browses source code relative to a particular file, procedure, or line.
- *dprint*: Evaluates an expression or program variable and displays the resulting value.
- *dptsets*: Shows status of processes and threads in a P/T set.
- *dstatus*: Shows status of processes and threads.
- *dup*: Navigates through the call stack by manipulating the current frame.
- *dwhat*: Determines what a name refers to.
- *dwhere*: Prints information about the thread's stack.

Execution Control

The following commands control execution:

- *dcont*: Continues execution of processes and waits for them.
- *dfocus*: Changes the set of processes, threads, or groups upon which a CLI command acts.
- *dgo*: Resumes execution of processes (without blocking).
- *dgroups*: Manipulates and manages groups.
- *dhalt*: Suspends execution of processes.
- *dhold*: Holds threads or processes.
- *dnext*: Executes statements, stepping over subfunctions.
- *dnexti*: Executes machine instructions, stepping over subfunctions.
- *dout*: Runs out of current procedure.
- *dstep*: Executes statements, moving into subfunctions if required.
- *dstepi*: Executes machine instructions, moving into subfunctions if required.
- *dunhold*: Releases held threads.
- *duntil*: Executes statements until a statement is reached.
- *dwait*: Blocks command input until processes stop.
- *dworker*: Adds or removes threads from a workers group.

Action Points

The following action point commands are responsible for defining and manipulating the points at which the flow of program execution should stop so that you can examine debugger or program state:

- *dactions*: Views information on action point definitions and their current status; it also saves and restores action points.
- *dbarrier*: Defines a process barrier breakpoint.
- *dbreak*: Defines a breakpoint.
- *ddelete*: Deletes an action point.
- *ddisable*: temporarily disables an action point.
- *denable*: Reenables an action point that has been disabled.
- *dwatch*: Defines a watchpoint.

Other Commands

The commands in the category do not fit into any of the other categories.

- *dassign*: Changes the value of a scalar variable.
- *dcheckpoint*: Creates a file that can later be used to restart a program.
- *drestart*: Restarts a checkpoint.

Accessor Functions

The following functions, all within the **TV::** namespace, access and set TotalView properties:

- *actionpoint*: Accesses and sets action point properties.
- *focus_groups*: Returns a list containing the groups in the current focus.
- *focus_processes*: Returns a list of processes in the current focus.
- *focus_threads*: Returns a list of threads in the current focus.
- *group*: Accesses and sets group properties.
- *image*: Accesses and sets image properties.
- *process*: Accesses and sets process properties.
- *prototype*: Accesses and sets prototype properties.
- *thread*: Accesses and sets thread properties.
- *type*: Accesses and sets data type properties.

Helper Functions

The following commands, all within the **TV::** namespace, are most often used within scripts:

- *dec2hex*: Converts a decimal number into hexadecimal format.
- *dlappend*: Appends list elements to a TotalView variable.
- *errorCodes*: Returns or raises TotalView error information.
- *hex2dec*: Converts a hexadecimal number into decimal format.
- *respond*: Sends a response to a command.
- *source_process_startup*: "Sources" a **.tvd** file when a process is loaded

actionpoint

Sets and gets action point properties

Format:

TV::**actionpoint** *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Lists the subcommands that you can use. The CLI responds by displaying the four subcommands shown here. No other arguments are used with this subcommand.
- **get** — Retrieves the values of one or more action point properties. *other-args* can include one or more property names. The CLI returns values for these properties in a list whose order is the same as the property names you entered.

  If you use the **–all** option as the *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists the action point properties that TotalView can access. No other arguments are used with this subcommand.
- **set** — Sets the values of one or more properties. *other-args* contains property name and value pairs.

object-id — An identifier for the action point.

other-args — Are arguments that the **get** and **set** actions use.

Description:

The **TV::actionpoint** command lets you examine and set action point properties and states. These states and properties are:

- **address** — The address of the action point.
- **enabled** — A value (either 1 or 0) indicating if the action point is enabled. 1 means enabled. (settable)
- **expression** — The expression to be executed at an action point. (settable)
- **id** — The ID of the action point.

language	The language in which the action point expression is written.
length	The length in bytes of a watched area. This property is only valid for watchpoints. (settable)
line	The source line at which the action point is set. This is not valid for watchpoints.
satisfaction_group	The group that must arrive at a barrier for the barrier to be *satisfied*. (settable)
share	A value (either 1 or 0) indicating if the action point is active in the entire share group. 1 means that it is. (settable)
stop_when_done	Indicates what is stopped when a barrier is satisfied (in addition to the satisfaction set). Values are **process**, **group**, or **none**. (settable)
stop_when_hit	Indicates what is stopped when an action point is hit (in addition to the thread that hit the action point). Values are **process**, **group**, or **none**. (settable)
type	The object's type. See **type_values** for a list of possible types.
type_values	Lists values that can be assigned to the **type** property: **break**, **eval**, **process_barrier**, **thread_barrier**, and **watch**.

Examples:

```
TV::actionpoint set 5 share 1 enable 1
```
Shares and enables action point 5.

```
f p3 TV::actionpoint set –all enable 0
```
Disables all the action points in process 3.

```
foreach p [TV::actionpoint properties] {
    puts [format "%20s %s" $p: [TV::actionpoint get 1 $p]]
```
Dumps all the properties for action point 1. Here is what your output might look like:

```
             address: 0x1200019a8
             enabled: 0
          expression:
                  id: 1
            language:
              length:
```

```
                line: /temp/arrays.F#84
satisfaction_group:
satisfaction_process:
satisfaction_width:
               share: 1
      stop_when_done:
        stop_when_hit: group
                type: break
         type_values: break eval process_barrier
                      thread_barrier watch
```

alias

Creates or views pseudonyms for commands

Format:

Creates a new user-defined pseudonym for a command

alias *alias-name defn-body*

Views previously defined aliases

alias [*alias-name*]

Arguments:

alias-name	The name of the command pseudonym being defined.
defn-body	The text that Tcl will substitute when it encounters *alias-name*.

Description:

The **alias** command associates a name you specify with text that you define. This text can contain one or more commands. After you create an alias, you can use it in the same way as a native TotalView or Tcl command. In addition, you can include an alias as part of a definition of another alias.

If you just do not enter an *alias-name* argument, the CLI displays the names and definitions of all aliases. If you just specify an *alias-name* argument, the CLI displays the definition of the alias.

Because the **alias** command can contain Tcl commands, you must ensure that *defn-body* complies with all Tcl expansion, substitution, and quoting rules.

TotalView's global startup file, **tvdinit.tvd**, defines a set of default aliases. All the common commands have one- or two-letter aliases. (You can obtain a list of these commands by typing **alias**—being sure not to use an argument—in the CLI window.)

You cannot use an alias to redefine the name of a CLI-defined command. You can, however, redefine a built-in CLI command by creating your own Tcl procedure. For example, here is a procedure that disables the built-in **dwatch** command. When a user types **dwatch**, the CLI executes this code instead of the built-in CLI code:

```
proc dwatch {} {
    puts "The dwatch command is disabled"
}
```

The CLI does not parse *defn-body* (the command's definition) until it is used. Thus, you can create aliases that are nonsensical or incorrect. The CLI only detects errors when it tries to execute your alias.

When you obtain help for a command, the help text includes information for TotalView's predefined aliases.

Examples:

alias nt dnext Defines a command called **nt** that executes the **dnext** command.

alias nt Displays the definition of the **nt** alias.

alias Displays the definitions of all aliases.

alias m {dlist main}
Defines an alias called **m** that lists the source code of function **main**.

alias step2 {dstep; dstep}
Defines an alias called **step2** that does two **dstep** commands. This new command will apply to the focus that exists when someone uses this alias.

alias step2 {s ; s} Creates an alias that performs the same operations as the one in the previous example. It differs in that it uses the alias for **dstep**. Note that you could also create an alias that does the same thing as follows: **alias step2 {s 2}**.

alias step1 {f p1. dstep}
Defines an alias called **step1** that steps the first user thread in process 1. Note that all other threads in the process run freely while TotalView steps the current line in your program.

capture

Returns a command's output as a string

Format:

capture *command*

Arguments:

command — The CLI command (or commands) whose output is being captured. If you are specifying more than one command, you must enclose them within braces ({ }).

Description:

The **capture** command executes *command*, capturing all output that would normally go to the console into a string. After *command* completes, it returns the string. The **capture** command lets you obtain the printed output of any CLI command so that you can assign it to a variable or otherwise manipulate it. This command is analogous to the UNIX shell's back-tick feature; that is, `*command*`.

Examples:

set save_stat [capture st]

Saves the current process status into a Tcl variable.

set vbl [capture {foreach i {1 2 3 4} {p int2_array($i)}}]

Saves the printed output of four array elements into a Tcl variable. Here is some sample output:

```
int2_array(1) = -8 (0xfff8)
int2_array(2) = -6 (0xfffa)
int2_array(3) = -4 (0xfffc)
int2_array(4) = -2 (0xfffe)
```

Because **capture** records all of the information sent to it by the commands in the foreach, you do not have to use a **dlist** command.

exec cat << [capture help commands] > cli_help.txt

Writes the help text for all TotalView commands to the **cli_help.txt** file.

dactions

Displays information, saves, and reloads action points

Format:

Displays information about action points

dactions [*ap-id-list*] [**–at** *source-loc*] [**–enabled** | **–disabled**]

Saves action points to a file

dactions –save [*filename*]

Loads previously saved action points

dactions –load [*filename*]

Arguments:

ap-id-list	A list of action point identifiers. If you specify individual action points, the information displayed is limited to these points. If you omit this argument, TotalView displays summary information about all action points in the processes in the focus set. If one ID is entered, TotalView displays full information for it. If more than one ID is entered, TotalView just displays summary information for each.
–at *source-loc*	Displays the action points at *source-loc*.
–enabled	Only shows enabled action points.
–disabled	Only shows disabled action points.
–save	Writes information about action points to a file.
–load	Restores action point information previously saved in a file.
filename	The name of the file into which TotalView will read and write action point information. If you omit this filename, TotalView writes them to a file named *program_name*.**TVD.breakpoints**, where *program_name* is the name of your program.

Description:

The **dactions** command displays information about action points in the processes in the current focus. The information is printed; it is not returned.

This command also lets you obtain the action point identifier. You will need to use this identifier when you delete, enable, and disable action points.

NOTE The identifier is returned when the action point is created. It is also displayed when the target stops at an action point.

You can include specific action point identifiers as arguments to the command when detailed information is required. The **–enabled** and **–disabled** options restrict output to action points in one of these states.

You cannot use the **dactions** command when you are debugging a core file or before TotalView loads executables.

The **–save** option tells TotalView that it should write action point information to a file so that either you or TotalView can restore your action points at a later time. The **–load** option tells TotalView that it should immediately read in the saved file. If you use the *filename* argument with either of these options, TotalView either writes to or reads from this file. If you do not use this argument it uses a file named *programname*.**TVD.breakpoints** where *programname* is the name of your program. This file is written to the same directory as your program.

The information saved includes expression information associated with the action point and whether the action point is enabled or disabled. For example, if your program's name is **foo**, it writes this information to **foo.TVD.breakpoints**.

NOTE TotalView does not save information about watchpoints.

If a file with the default name exists, TotalView can read this information when it starts your program. When TotalView exits, it can create the default. For more information, see **File > Preferences** within TotalView's Help system.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
ac	{dactions}	Displays all action points

Examples:

ac –at 81

Displays information about the action points on line 81. (Notice that this example uses the alias instead of the full command name.) Here is the output from this command:

```
ac –at 81
1 shared action point for group 3:
  1 addr=0x10001544 [arrays.F#81] Enabled
    Share in group: true
    Stop when hit: group
```

dactions 1 3

Displays information about action points 1 and 3, as follows:

```
2 shared action points for process 1:
    1 addr=0x100012a8 [arrays.F#56] Enabled
    3 addr=0x100012c0 [arrays.F#57] Enabled
```

dfocus p1 dactions

Displays information on all action points defined within process 1.

dfocus p1 dactions –enabled

Displays information on all enabled action points within process 1.

dassign

Changes the value of a scalar variable

Format:

dassign *target value*

Arguments:

target	The name of a scalar variable within your program.
value	A source-language expression that evaluates to a scalar value. This expression can use the name of another variable.

Description:

The **dassign** command evaluates an expression and replaces the value of a variable with the evaluated result. The location may be a scalar variable, a dereferenced pointer variable, or an element in an array or structure.

The default focus for **dassign** is *thread*. So, if you do not change the focus, this command acts upon the *thread of interest*. If the current focus specifies a width that is wider than **t** (thread) and is not **d** (default), **dassign** iterates over the threads in the focus set and performs the assignment in each. In addition, if you use a list with the **dfocus** command, **dassign** iterates over each list member.

The CLI interprets each symbol name in the expression according to the current context. Because the value of a source variable may not have the same value across threads and processes, the value assigned can differ in your threads and processes. If the data type of the resulting value is incompatible with that of the target location, you must cast the value into the target's type. (*Casting* is described in Chapter 7 of the TOTALVIEW USERS GUIDE.)

Here are some things you should know about assigning characters and strings:

- If you are assigning a character to a *target*, place the character value within single quotation marks; for example, **'c'**.
- You can use the standard C language escape character sequences; for example, **\n**, **\t**, and the like. These escape sequences can also be within a character or string assignment.

- If you are assigning a string to a *target*, place the string within quotation marks. However, you must "escape" the quotation marks so they are not interpreted by Tcl; for example, **\"The quick brown fox\"**.

If *value* contains an expression, the expression is evaluated by TotalView's expression system. This system is discussed in Chapter 8 of the TOTALVIEW USERS GUIDE.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
as	{dassign}	Changes a scalar variable's value.

Examples:

dassign scalar_y 102

Stores the value 102 in each occurrence of variable **scalar_y** for all processes and threads in the current set.

dassign i 10*10 Stores the value 100 in variable **i**.

dassign i i*i Does not work and the CLI displays an error message. If **i** is a simple scalar variable, you could use the following statements:

```
set x [lindex [capture dprint i] 2]
dassign i [expr $x * $x]
```

f {p1 p2 p3} as scalar_y 102

Stores the value 102 in each occurrence of variable **scalar_y** contained within processes 1, 2, and 3.

dattach

Brings currently executing processes under CLI control

Format:

dattach [–g *gid*] [–r *hname*]
[–ask_attach_parallel | –no_attach_parallel]
[–e] *fname pid-list*

Arguments:

–g *gid* — Sets the control group for the processes being added to be group *gid*. This group must already exist. (The CLI **GROUPS** variable contains a list of all groups. See **GROUPS** on page 203 for more information.)

–r *hname* — The host on which the process is running. The CLI will launch a TotalView Debugger Server on the host machine if one is not already running there. Consult the TOTALVIEW USER GUIDE for information on the launch command used to start this server.

Setting a host sets it for all PIDs attached to in this command. If you do not name a host machine, the CLI uses the local host.

–ask_attach_parallel — Asks if TotalView should attach to parallel processes of a parallel job. The default is to automatically attach to processes. For additional information, see **File > Preferences** in TotalView's Help.

–no_attach_parallel — Do not attach to any additional parallel processes within a parallel job. For additional information, see **File > Preferences** in TotalView's Help.

–e — Tells the CLI that the next argument is a file name. You need to use this argument if the file name begins with a dash (–) or only uses numeric characters.

fname — The name of the executable. Setting an executable here, sets it for all PIDs being attached to in this command. If you do not include this argument, the CLI tries to determine the executable file from the process. Some architectures do not allow this to occur.

pid-list — A list of system-level process identifiers (such as a UNIX PID) naming the processes that TotalView will

control. All PIDs must reside on the same system and they will all be placed into the same control group.

If you need to place the processes in different groups or attach to processes on more than one system, you must use multiple **dattach** commands.

Description:

The **dattach** command tells TotalView to attach to one or more processes, making it possible to continue process execution under CLI control.

This command returns the TotalView process ID (DPID) as a string. If you specify more than one process in a command, **dattach** returns a list of DPIDs instead of a single value.

TotalView places all processes to which it attaches in one **dattach** command in the same control group. This allows you to place all processes in a multiprocess program executing on the same system in the same control group.

If a program has more than one executable, you must use a separate **dattach** for each.

If the *fname* executable is not already loaded, the CLI searches for it. The search will include all directories in the **EXECUTABLE_PATH** CLI state variable.

The process identifiers specified in the *pid-list* must refer to existing processes in the run-time environment. TotalView attaches to the processes, regardless of their execution states.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
at	{dattach}	Brings the process under CLI control

Examples:

dattach mysys 10020

Loads debugging information for **mysys** and brings the process known to the run-time system by PID 10020 under CLI control.

dattach –e 123 10020

Loads file 123 and brings the process known to the run-time system by PID 10020 under CLI control.

dattach –g 4 –r Enterprise myfile 10020

Loads **myfile** that is executing on the host named **Enterprise** into group 4 and brings the process known to the run-time system by PID 10020 under CLI control. If a TotalView Debugger Server (**tvdsvr**) is not running on **Enterprise**, the CLI will start it.

dattach my_file 51172 52006

Loads debugging information for **my_file** and brings the processes corresponding to PIDs 51172 and 52006 under CLI control.

```
set new_pid [dattach –e mainprog 123]
dattach –r otherhost –g $CGROUP($new_pid) –e slaveprog 456
```

Begins by attaching to **mainprog** running on the local host. It then attaches to **slaveprog** running on **otherhost** and inserts them both in the same control group.

dbarrier

Defines a process or thread barrier breakpoint

Format:

Creates a barrier breakpoint at a source location

dbarrier *source-loc* [**–stop_when_hit** *width*] [**–stop_when_done** *width*]

Creates a barrier breakpoint at an address

dbarrier –address *addr* [**–stop_when_hit** *width*] [**–stop_when_done** *width*]

Arguments:

source-loc — The breakpoint location as a line number or as a string containing a file name, function name, and line number, each separated by **#** characters; for example, **#file#line**. If you omit parts of this specification, the CLI will create them for you. For more information, see "Qualifying Symbol Names" on page 80.

–address *addr* — The breakpoint location as an absolute address in the address space of the program.

–stop_when_hit *width*

Tells the CLI what else it should stop when it stops the thread arriving at a barrier.

If you do not use this option, the value of the **BARRIER_STOP_ALL** state variable indicates what TotalView will stop.

This command's *width* argument indicates what else TotalView stops. You can enter one of the following three values:

group — Stops all processes in the control group when the barrier is hit.

process — Stops the process that hit the barrier.

none — Stops the thread that hit the barrier; that is, the thread will be held and all other threads continue running. If you apply this width to a process barrier, TotalView will stop the process that hit the breakpoint.

–stop_when_done *width*

After all processes or threads reach the barrier, the CLI releases all processes and threads held at the barrier. (*Released* means that these threads and processes can run.) Setting this option tells the CLI that it should stop additional threads contained in the same **group** or **process**.

If you do not use this option, the value of the **BARRIER_STOP_WHEN_DONE** state variable indicates what else TotalView stops.

The *width* argument indicates what else is stopped. You can enter one of the following three values:

group	Stops the entire control group when the barrier is satisfied.
process	Stops the processes containing threads in the satisfaction set when the barrier is satisfied. You will find information on the "satisfaction set" in this topic's *Description* section.
none	Stops the satisfaction set. For process barriers, **process** and **none** have the same effect. This is the default if **BARRIER_STOP_WHEN_DONE** is **none**.

Description:

The **dbarrier** command sets a process or thread barrier breakpoint that is triggered when execution arrives at a location. This command returns the ID of the newly created breakpoint.

The **dbarrier** command is most often used to synchronize a set of threads. The P/T set defines which threads are affected by the barrier. When a thread reaches a barrier, it stops, just as it does for a breakpoint. The difference is that TotalView prevents—that is, holds—each thread reaching the barrier from responding to resume commands (for example, **dstep**, **dnext**, and **dgo**) until all threads in the affected set arrive at the barrier. When all threads reach the barrier, TotalView considers the barrier to be *satisfied* and releases these threads. T*hey are just released*; *they are not continued*. That is, TotalView leaves them stopped at the barrier. If you now continue the process, those threads stopped at the barrier also run along with any other

threads that were not participating with the barrier. After they are released, they can respond to resume commands.

If the process is stopped and then continued, the held threads, including the ones waiting on an unsatisifed barrier, do not run. Only unheld threads run.

The satisfaction set for the barrier is determined by the current focus. If the focus group is a thread group, TotalView creates a thread barrier.

- When a thread hits a process barrier, TotalView holds the thread's process.
- When a thread hits a thread barrier, TotalView holds the thread; TotalView may also stop the thread's process or control group. Neither are held.

TotalView determines what processes and threads are part of the satisfaction set by taking the intersection of the share group with the focus set. (Barriers cannot extend beyond a share group.)

The CLI displays an error message if you use an inconsistent focus list.

NOTE Barriers can create deadlocks. For example, if two threads participate in two different barriers, each could be left waiting at different barriers, barriers that can never be satisfied. A deadlock can also occur if a barrier is set in a procedure that will never be invoked by a thread in the affected set. If a deadlock occurs, use the ddelete command to remove the barrier since deleting the barrier also releases any threads held at the barrier.

The **–stop_when_hit** option tells TotalView what other threads it should stop when a thread arrives at a barrier.

The **–stop_when_done** option controls the set of additional threads that TotalView will stop when the barrier is finally satisfied. That is, you can also stop an additional collection of threads after the last expected thread arrives and all the threads held at the barrier are released, Normally, you will want to stop the threads contained in the control group.

If you omit a *stop* option, TotalView sets the default behavior by using the **BARRIER_STOP_ALL** and **BARRIER_STOP_WHEN_DONE** state variables. For more information, see **dset**.

The **none** argument for these options indicate that the CLI should not stop additional threads.

- If **–stop_when_hit** is **none** when a thread hits a thread barrier, TotalView just stops that thread; it does not stop other threads.
- If **–stop_when_done** to **none**, TotalView does not stop additional threads, aside from the ones that are already stopped at the barrier.

TotalView plants the barrier point in the processes or groups specified in the current focus. If the current focus:

- Does not indicate an explicit group, the CLI creates a process barrier across the share group.
- Indicates a process group, the CLI creates a process barrier that is satisfied when all members of that group reach the barrier.
- Indicates a thread group, TotalView creates a thread barrier that is satisfied when all members of the group arrive at the barrier.

The following example illustrates these differences. If you set a barrier with the focus set to a control group (which is the default), TotalView creates a process barrier. This means that the **–stop_when_hit** value is set to **process** even though you specified **thread**.

```
d1.<>  dbarrier 580 –stop_when_hit thread
2
d1.<> ac 2
1 shared action point for group 3:
  2 addr=0x120005598 [../regress/fork_loop.cxx#580]
Enabled (barrier)
   Share in group: true
   Stop when hit: process
   Stop when done: process
   process barrier; satisfaction set = group 1
```

However, if you create the barrier with a specific workers focus, **stop_when_hit** remains set to **thread**:

```
1.<> baw 580 –stop_when_hit thread
1
d1.<> ac 1
1 unshared action point for process 1:
  1 addr=0x120005598 [../regress/fork_loop.cxx#580]
                              Enabled (barrier)
```

```
Share in group: false
Stop when hit: thread
Stop when done: process
thread barrier; satisfaction set = group 2
```

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
ba	{dbarrier}	Defines a barrier.
baw	{dfocus pW dbarrier –stop_when_done process}	Creates a thread barrier across the worker threads in the process of interest. TotalView sets the set of threads stopped when the barrier is satisfied to the process containing the satisfaction set.
BAW	{dfocus gW dbarrier –stop_when_done group}	Creates a thread barrier across the worker threads in the share group of interest. The set of threads stopped when the barrier is satisfied will be the entire control group.

Examples:

dbarrier 123

Stops each process in the control group when it A arrives at line 123, it is stopped. After all arrive, the barrier is satisfied and TotalView releases all processes.

dfocus {p1 p2 p3} dbarrier my_proc

Holds each thread in processes 1, 2, and 3 as it arrives at the first executable line in procedure **my_proc**. After all arrive, the barrier is satisfied and TotalView releases all processes.

dfocus gW dbarrier 642 –stop_when_hit none

Sets a thread barrier at line 642 on the workers group. The process is continued automatically as each thread arrives at the barrier. That is, threads that are not at this line continue running.

dbreak

Defines a breakpoint

Format:

Creates a breakpoint at a source location

dbreak *source-loc* [**–p** | **–g** | **–t**] [[**–l** *lang*] **–e** *expr*]

Creates a breakpoint at an address

dbreak –address *addr* [**–p** | **–g** | **–t**] [[**–l** *lang*] **–e** *expr*]

Arguments:

source-loc	The breakpoint location specified as a line number or as a string containing a file name, function name, and line number, each separated by **#** characters; for example, **#file#line**. Defaults are constructed if you omit parts of this specification. For more information, see "Qualifying Symbol Names" on page 80.
–address *addr*	The breakpoint location specified as an absolute address in the address space of the program.
–p	Tells TotalView to stop the process that hit this breakpoint. You can set this option as the default by setting the **STOP_ALL** state variable to **process**. See **dset** on page 200 for more information.
–g	Tells TotalView to stop all processes in the process's control group when the breakpoint is hit. You can set this option as the default by setting the **STOP_ALL** state variable to **group**. See **dset** on page 200 for more information.
–t	Tells TotalView to stop the thread that hit this breakpoint. You can set this option as the default by setting the **STOP_ALL** state variable to **thread**. See **dset** on page 200 for more information.
–l *lang*	Sets the programming language used when you are entering expression *expr*. The languages you can enter are **c**, **c++**, **f7**, **f9**, and **asm** (for C, C++, FORTRAN 77, Fortran 9x, and assembler). If you do not specify a language, TotalView assumes that you wrote the expression in the same language as the routine at the breakpoint.

–**e** *expr*
When the breakpoint is hit, TotalView will evaluate expression *expr* in the context of the thread that hit the breakpoint. The language statements and operators you can use are described in the TOTALVIEW USERS GUIDE.

Description:

The **dbreak** command defines a breakpoint or evaluation point that TotalView triggers when execution arrives at the specified location. The ID of the new breakpoint is returned.

Each thread stops when it arrives at a breakpoint.

Specifying a procedure name without a line number tells the CLI to set an action point at the beginning of the procedure. If you do not name a file, the default is the file associated with the current source location.

The CLI may not be able to set a breakpoint at the line you specify. This occurs when a line does not contain an executable statement.

If you try to set a breakpoint on a line at which the CLI cannot stop execution, it sets one at the nearest following line where it can halt execution.

When the CLI displays information on a breakpoint's status, it displays the location where execution will actually stop.

If the CLI encounters a *stop group* breakpoint, it suspends each process in the group as well as the process containing the triggering thread. The CLI then shows the identifier of the triggering thread, the breakpoint location, and the action point identifier.

One possibly confusing aspect of using expressions is that their syntax differs from that of Tcl. This is because you will need to embed code written in Fortran, C, or assembler within Tcl commands. In addition, your expressions will often include TotalView intrinsic functions. For example, if you want to use the TotalView **$tid** built-in function, you will need to type it as **\$tid**.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
b	{break}	Sets a breakpoint.
bt	{dbreak t}	Sets a breakpoint just on the thread of interest.

Examples:

For all examples, assume the current process set is **d2.<** when the breakpoint is defined.

dbreak 12
Suspends process 2 when it reaches line 12. However, if the **STOP_ALL** state variable is set to **group**, all other processes in the group are stopped. In addition, if you have set the **SHARE_ACTION_POINT** state variable to **true**, the breakpoint is placed in every process in the group.

dbreak –address 0x1000764
Suspends process 2 when address 0x1000764 is reached.

b 12 –g
Suspends all processes in the current control group when line 12 is reached.

dbreak 57 –l f9 –e {goto $63}
Causes the thread that struck the breakpoint to transfer to line 63. The host language for this statement is Fortran 90 or Fortran 95.

dfocus p3 b 57 –e {goto $63}
In process 3, sets the same evaluation point as the previous example.

dcheckpoint

Creates a checkpoint image of processes (SGI only)

Format:

dcheckpoint [*after_checkpointing*] [–by *process_set*] [–no_park]
[–ask_attach_parallel | –no_attach_parallel]
[–no_preserve_ids] [–force] *checkpoint-name*

Arguments:

after_checkpointing	Defines the state of the process both before and after the checkpoint. Use one of the following options:
–delete	Processes exit after being checkpointed.
–detach	Processes continue running after being checkpointed. In addition, TotalView detaches from them.
–go	Processes continue running after being checkpointed.
–halt	Processes halt after they are checkpointed.
–by *process_set*	Indicates the set of processes that will be checkpointed. If you do not use a *process_set* option, TotalView only checkpoints the focus process. Your options are:
ash	Checkpoint the array session. (SGI only)
hid	Checkpoint the hierarchy rooted in the focus process.
pgid	Checkpoint the entire UNIX process group.
sid	Checkpoint the entire process session.
–no_park	Tells TotalView that it should not *park* all processes before TotalView begins checkpointing them. If you use this option, you will also need to use the **drestart** command's **–no_unpark** option. Checkpoints that will be restarted from a shell must use this option.
–ask_attach_parallel	Asks if TotalView should reattach to parallel processes of a parallel job. (Some systems automatically detach you from processes being checkpointed.)
–no_attach_parallel	Tells TotalView that it should not reattach to processes from which the checkpointing processes detached. (Some systems automatically detach you from processes being checkpointed.)

–no_preserve_ids	Lets TotalView assign new IDs when it restarts a checkpoint. If you do not use this option, the same IDs are used.
–force	Tells TotalView to overwrite an existing checkpoint.
checkpoint-name	The name being assigned to the checkpoint.

Description:

The **dcheckpoint** command saves program and process information. into the *checkpoint-name* file. This information includes process and group IDs. Some time later, you will use the **drestart** command to restart the program.

NOTE This command does not save TotalView breakpoint information.

The following restrictions exist when you are trying to checkpoint IRIX processes.

- IRIX will not checkpoint a process that is running remotely and which communicates using sockets. As the TotalView Server (**tvdsvr**) uses sockets to redirect **stdin**, **stdout**, and **stderr**, you will need to used the **drun** command to modify the way your processes send information to a **tty** before creating a checkpoint.
- Because SGI MPI makes extensive use of sockets, you cannot checkpoint SGI MPI programs.

The *after_checkpointing* options let you specify what happens after the checkpoint operation concludes. If you do not specify an option, the CLI tells the checkpointed processes that they should stop. This lets you investigate a program's state at the checkpoint position. In contrast, **–go** tells the CLI that it should let the processes continue to run. The **–detach** and **–halt** options are used less frequently. The **–detach** option shuts down the CLI and leaves the processes running. The **–halt** option is similar to **–detach**, differing only in that processes started by the CLI and TotalView are also terminated.

The *process_set* options tell TotalView which processes it should checkpoint. While the focus set can only contain one process, processes within the same process group, process session, process hierarchy, or array session can also be included within the same checkpoint. If you do not use one of the **–by** options, TotalView only checkpoints the focus process.

If the focus group contains more than one process, the CLI displays an error message.

Just before TotalView begins checkpointing your program, it temporarily stops (that is, *parks*) the processes that being checkpointed. Parking ensures that the processes do not run freely after a **dcheckpoint** or **drestart** operation. (If they did, your code would begin running before you get control of it.) If you will be restarting the checkpoint file outside of TotalView, you must use the **–no_park** option.

On some operating systems (including SGI), the CLI detaches from processes before they are checkpointed. By default, the CLI automatically reattaches to them. If you want something different to occur, you can tell the CLI that it should never reattach (**–no_attach_parallel**) or that it should ask you if it should reattach (**–ask_attach_parallel**).

Examples:

dcheckpoint check1

Checkpoints the current process. TotalView writes the checkpoint information into the *check*1 file. These processes stop.

f3 dcheckpoint check1

Checkpoints process 3. Process 3 stops. TotalView writes the checkpoint information into the *check*1 file.

f3 dcheckpoint –go check1

Checkpoints process 3. Process 3 continues to run. TotalView writes the checkpoint information is into the *check*1 file.

f3 dcheckpoint –by pgid –detach check1

Checkpoints process 3 and all other processes in the same UNIX process group. All of the checkpointed processes continuing running but they run detached from the CLI. TotalView writes the checkpoint information into the *check*1 file.

dcont

Continues execution and waits for execution to stop

Format:

dcont

Description:

The **dcont** command continues all processes and threads in the current focus and then waits for all of them to stop.

This command is a Tcl macro whose definition is as follows:

```
proc dcont {args} {uplevel "dgo; dwait $args"}
```

This behavior is often what you want to do in scripts. It is seldom what you want to do interactively.

NOTE You can interrupt this action by typing Ctrl-C. This tells TotalView to stop executing these processes.

A **dcont** command completes when all threads in the focus set of processes stop executing.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
co	{dcont}	Resume.
CO	{dfocus g dcont}	Group-level resume.

Examples:

dcont — Resumes execution of all *stopped/runnable* threads belonging to processes in the current focus. (Threads held at barriers are not affected.) The command blocks further input until all threads in all target processes stop. After the CLI displays its prompt, you can enter additional commands.

dfocus p1 dcont — Resumes execution of all *stopped/runnable* threads belonging to process 1. The CLI does not accept additional commands until the process stops.

`dfocus {p1 p2 p3} co`
: Resumes execution of all *stopped/runnable* threads belonging to processes 1, 2, and 3.

`CO`
: Resumes execution of all *stopped/runnable* threads belonging to the current group.

ddelete

Deletes action points

Format:

Deletes some action points

ddelete *action-point-list*

Deletes all action points

ddelete –a

Arguments:

action-point-list	A list of the action points being deleted.
–a	Tells TotalView to delete all action points in the current focus.

Description:

The **ddelete** command permanently removes one or more action points. The argument to this command lets you specify which action points the CLI should delete. The **–a** option indicates that the CLI should delete all action points.

If you delete a barrier point, the CLI releases the processes and threads held at it.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
de	{ddelete}	Deletes action points.

Examples:

ddelete 1 2 3	Deletes breakpoints 1, 2, and 3.
ddelete –a	Deletes all action points associated with processes in the current focus.
dfocus {p1 p2 p3 p4} ddelete –a	Deletes all the breakpoints associated with processes 1 through 4. Breakpoints associated with other threads are not affected.
dfocus a de –a	Deletes all action points known to the CLI.

ddetach

Detaches from processes

Format:

ddetach

Description:

The **ddetach** command detaches the CLI from all processes in the current focus. This *undoes* the effects of attaching the CLI to a running process; that is, the CLI releases all control over the process, eliminates all debugger state information related to it (including action points), and allows the process to continue executing in the normal run-time environment.

You can detach any process controlled by the CLI; the process being detached does not have to be originally loaded with a **dattach** command.

After this command executes, you are no longer able to access program variables, source location, action point settings, or other information related to the detached process.

If a single thread serves as the set, the CLI detaches the process containing the thread.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
det	{ddetach}	Detaches from processes.

Examples:

ddetach Detaches the process or processes that are in the current focus.

dfocus {p4 p5 p6} det
Detaches processes 4, 5, and 6.

dfocus g2 det Detaches all processes in the control group associated with process 2.

ddisable Temporarily disables action points

Format:

Disables some action points

ddisable *action-point-list*

Disables all action points

ddisable –a

Arguments:

action-point-list	A list of the action points being disabled.
–a	Tells TotalView to disable all action points.

Description:

The **ddisable** command temporarily deactivates action points. This command does not, however, delete them.

The first form of this command lets you explicitly name the IDs of the action points being disabled. The second form lets you disable all action points.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
di	{ddisable}	Temporarily disables action points

Examples:

ddisable 3 7	Disables the action points whose IDs are 3 and 7.
di –a	Disables all action points in the current focus.
dfocus {p1 p2 p3 p4} ddisable –a	Disables action points associated with processes 1 through 4. Action points associated with other processes are not affected.

ddown

Moves down the call stack

Format:

ddown [*num-levels*]

Arguments:

num-levels	Number of levels to move down. The default is 1.

Description:

The **ddown** command moves the selected stack frame down one or more levels. It also prints the new frame's number and function name.

Call stack movements are all relative, so **ddown** effectively "moves down" in the call stack. (If "up" is in the direction of **main()**, then "down" is back from where you started moving through stack frames.)

Frame 0 is the most recent—that is, the currently executing—frame in the call stack, frame 1 corresponds to the procedure that invoked the currently executing one, and so on. The call stack's depth is increased by one each time a procedure is entered, and decreased by one when it is exited.

The command affects each thread in the focus. You can specify any collection of processes and threads as the target set.

In addition, the **ddown** command modifies the current list location to be the current execution location for the new frame; this means that a **dlist** command displays the code surrounding this new location.

The context and scope changes made by this command remain in effect until the CLI executes a command that modifies the current execution location (for example, **dstep**), or until you enter a **dup** or **ddown** command.

If you tell the CLI to move down more levels than exist, the CLI simply moves down to the lowest level in the stack (which was the place where you began moving through the stack frames).

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
d	{ddown}	Moves down the call stack

Examples:

ddown

Moves down one level in the call stack. As a result, for example, **dlist** commands that follow will refer to the procedure that invoked this one. Here is an example of what is printed after you enter this command:

```
0 check_fortran_arrays_ PC=0x10001254,
    FP=0x7fff2ed0 [arrays.F#48]
```

d 5

Moves the current frame down five levels in the call stack.

dec2hex

Converts a decimal number into hexadecimal

Format:

TV::dec2hex *number*

Arguments:

number	A decimal number.

Description:

Converts a decimal number into hexadecimal. This command correctly manipulates 64-bit values, regardless of the size of a **long** on the host system.

denable Enables action points

Format:

Enables some action points

denable *action-point-list*

Enables all disabled action points in the current focus

denable –a

Arguments:

action-point-list	The identifiers of the action points being enabled.
–a	Tells TotalView to enable all action points.

Description:

The **denable** command reactivates action points that you had previously disabled with the **ddisable** command. The **–a** option tells the CLI to enable all action points in the current focus.

If you have not saved the ID values of disabled action points, you can use the **dactions** command to obtain a list of this information.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
en	{denable}	Reenables action points

Examples:

denable 3 4 Enables two previously identified action points. These action points were previously disabled with the **ddisable** command.

dfocus {p1 p2} denable –a Enables all action points associated with processes 1 and 2. Settings associated with other processes are not affected.

en –a Enables all action points associated with the current focus.

f a en –a Enables all actions points in all processes.

dfocus

Changes the current Process/Thread set

Format:

Changes the target of future CLI commands to this P/T set

dfocus *p/t-set*

Executes a command in this P/T set

dfocus *p/t-set command*

Arguments:

p/t-set	A set of processes and threads. This set defines the target upon which the CLI commands that follow will act. You can also use a P/T set filter as one or more of the elements in this list.
command	A CLI command, which when it executes, operates upon its own local focus.

Description:

The **dfocus** command changes the set of processes, threads, and groups upon which a command will act. This command can change the focus for all commands that follow or just the command that immediately follows.

The **dfocus** command always expects a P/T value as its first argument. This value can either be a single arena specifier or a list of arena specifiers. The default focus is **d1.<**, which selects the first user thread. The **d** (for default) indicates that each CLI command is free to use its own default width.

If you enter an optional *command*, the focus is set temporarily, and the CLI executes *command* in the new focus. After *command* executes, the CLI restores focus to its original value. The *command* argument can be a single command or a list.

If you use a *command* argument, **dfocus** returns the result of the command. If you do not enter a command, **dfocus** returns the focus as a string value.

NOTE Instead of a P/T set, you can type a P/T set expression. These expressions are described in "P/T Set Expressions" on page 63.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
f	{dfocus}	Changes object upon which a command acts.

Examples:

dfocus g dgo Continues the TotalView group containing the focus process.

dfocus p3 {dhalt; dwhere}
Stops process 3 and displays backtraces for each of its threads.

dfocus 2.3 Sets the focus to thread 3 of process 2, where the "2" and the "3" are TotalView's process and thread identifier values. The focus is set to **d2.3**.

dfocus 3.2
dfocus .5 Sets, then resets command focus. A focus command that includes a dot and omits the process value tells the CLI to use the current process. Thus, this sequence of commands changes the focus to *process* 3, *thread* 5 (**d3.5**).

dfocus g dstep Steps the current group. Note that while the thread of interest is determined by the current focus, the command acts on the entire group containing that thread.

dfocus {p2 p3} {dwhere ; dgo}
Performs a backtrace on all threads in processes 2 and 3 and then tells these processes to execute.

f 2.3 {f p w; f t s; g}
Executes a backtrace (**dwhere**) on all the threads in process 2, steps thread 3 in process 2 (without running any other threads in the process), and continues the process.

dfocus p1 Changes the current focus to include just those threads currently in process 1. The width is set to **process**. The CLI sets the prompt to **p1.<**.

dfocus a Changes the current set to include all threads in all processes. When you execute this command, you will notice that your prompt changes to **a1.<**. This command alters the CLI's behavior so that actions that previously operated on a thread now apply to all threads in all processes.

dfocus gW dstatus Displays the status of all worker threads in the control group. The width is group level and the target is the workers group.

dfocus pW dstatus Displays the status of all worker threads in the current focus process. The width is process level and the target is the workers group.

f {breakpoint(a) | watchpoint(a)} st
Shows all threads that are stopped at breakpoints or watchpoints.

f {stopped(a) – breakpoint(a)} st
Shows all stopped threads that are not stopped at breakpoints.

You will find many other **dfocus** examples in Chapter 3, "Groups, Processes, and Threads" on page 15.

dgo

Resumes execution of processes

Format:

dgo

Description:

The **dgo** command tells all non-held processes and threads in the current focus to resume execution. If the process does not exist, this command creates it, passing it the default command arguments. These can be arguments passed into the CLI or they can be the arguments set with the **drerun** command. If you are also using the TotalView GUI, this value can be set by using the **Process** > **Startup** command.

This command has no arguments.

If a process or thread is held, it ignores this command.

You cannot use a **dgo** command when you are debugging a core file, nor can you use it before the CLI loads an executable and starts executing it.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
g	{dgo}	Resumes execution.
G	{dfocus g dgo}	Group resume.

Examples:

dgo	Resumes execution of all *stopped/runnable* threads belonging to processes in the current focus. (Threads held at barriers are not affected.)
G	Resumes execution of all threads in the current control group.
f p g	Continues the current process. Only threads that are not held are actually allowed to run.
f g g	Continues all processes in the control group. Only processes and threads that are not held are allowed to run.
f gL g	Continues all threads in the share group that are at the same PC as the thread of interest.

f pL g	Continues all threads in the current process that are at the same PC as the thread of interest.
f t g	Continues a single thread.

dgroups

Manipulates and manages groups

Format:

Adds members to thread and process groups

dgroups –add [**–g** *gid*] [*id-list*]

Deletes groups

dgroups –delete [**–g** *gid*]

Intersects a group with a list of processes and threads

dgroups –intersect [**–g** *gid*] [*id-list*]

Prints process and thread group information

dgroups [**–list**] [*pattern-list*]

Creates a new thread or process group

dgroups –new [*thread_or_process*] [**–g** *gid*] [*id-list*]

Removes members from thread or process groups

dgroups –remove [**–g** *gid*] [*id-list*]

Arguments:

–g *gid*	The group ID upon which the command operates. *gid* can be an existing numeric group ID, an existing group name, or, if you are using the **–new** option, a new group name.
id-list	A Tcl list containing process and thread IDs. Process IDs are integers; for example, "2" indicates process 2. Thread IDs define a *pid.tid* pair and look like decimal numbers; for example, "2.3" indicates process 2, thread 3. If the first element of this list is a group tag such as the word **control**, the CLI ignores it. This makes it easy to insert all members of an existing group as the items to be used in any of these operations. (See the **dset** command's discussion of the **GROUP(gid)** variable for information on group designators.) These words appear in some circumstances when TotalView returns lists of elements in P/T sets.
thread_or_process	Keywords indicating that TotalView will create a new process or thread group. You can specify one of the following arguments: **t**, **thread**, **p**, or **process**.

pattern-list A pattern to be matched against group names. The pattern is a Tcl regular expression

Description:

The **dgroups** command lets you perform the following functions:

- Adds members to process and thread groups.
- Creates a group.
- Intersects a group with a set of processes and threads.
- Deletes groups.
- Displays the name and contents of groups.
- Removes members from a group.

dgroups –add

The **dgroups –add** command adds members to one or more thread or process groups. TotalView adds each of these threads and processes to the group. If you add a:

- Process to a thread group, TotalView adds all of its threads.
- Thread to a process group, it adds the thread's parent process.

You can abbreviate **–add** to **–a**.

The CLI returns the ID of this group.

The items being added can be explicitly named using an *id-list*. If you do not use an *id-list*, the CLI adds the threads and processes in the current focus. Similarly, you can name the group to which the CLI adds members if you use the **–g** option. If you omit this option, the CLI uses the groups in the current focus.

If *id-list* contains processes and the target is a thread group, the CLI adds all threads from these processes. If it contains threads and the target is a process group, TotalView adds the parent process for each thread.

NOTE If you specify an *id-list* and use the –g option, the CLI ignores the focus.

Even if you try to add the same object more than once to a group, the CLI only adds it once.

TotalView does not let you use this command to add a process to a control group. If you need to perform this operation, you can add it by using the **CGROUP(dpid)** variable. For example:

```
dset CGROUP($mypid) $new_group_id
```

dgroups –delete

The **dgroups –delete** command deletes the target group. You can only delete groups that you create; you cannot delete groups that TotalView creates.

dgroups –intersect

The **dgroups –intersect** command intersects a group with a set of processes and threads. If you intersect a thread group with a process, the CLI uses all of the process's threads. If you intersect a process group with a thread, the CLI uses the thread's process.

After this command executes, the group no longer contains members that were not in this intersection.

You can abbreviate **–intersect** to **–i**.

dgroups –list

The **dgroups –list** command prints the name and contents of process and thread groups. If you specify a *pattern-list* as an argument, the CLI only prints information about groups whose names match this pattern.

When entering a list, you can specify a *pattern*. The CLI matches this pattern against TotalView's list of group names by using the Tcl **regex** command.

NOTE If you do not enter a pattern, the CLI only displays groups that you have created which have nonnumeric names.

The CLI returns information from this command; it is not returned.

You can abbreviate **–list** to **–l**.

dgroups –new

The **dgroups –new** command creates a new thread or process group and adds threads and processes to it. If you use a name with **–g**, the CLI uses that name for the group ID; otherwise, it assigns a new numeric ID. If the group you name already exists, the CLI replaces it with the newly created group.

The CLI returns the ID of the newly created group.

The items being added can be explicitly named using an *id-list*. If you do not use an *id-list*, the CLI adds the threads and processes in the current focus.

If *id-list* contains processes and the target is a thread group, the CLI adds all threads from these processes. If it contains threads and the target is a process group, TotalView adds the parent process for each thread.

NOTE If you specify an *id-list* and use the –g option, the CLI ignores the focus.

If you are adding more than one object and one of these objects is a duplicate, TotalView will add the non-duplicate objects to the group.

You can abbreviate **–new** to **–n**.

dgroups –remove

The **dgroups –remove** command removes members from one or more thread or process groups. If you ask to remove a process from a thread group, TotalView removes all of its threads. If you ask to remove a thread from a process group, TotalView removes its parent process.

You cannot remove processes from a control group. You can, however, move a process from one control group to another by using the **dset** command to assign it to the **CGROUP(dpid)** variable group.

Also, you cannot use this command on readonly groups such as share groups.

You can abbreviate **–remove** to **–r**.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
gr	dgroups	Manipulates a group.

Examples:

dgroups –add

f tW gr –add
: Adds the focus thread to its workers group.

dgroups –add
: Adds the current focus thread to the current focus group.

set gid [dgroups –new thread ($CGROUP(1))]
: Creates a new thread group containing all threads from all processes in the control group for process 1.

f $a_group/9 dgroups –add
: Adds process 9 to a user-defined group.

dgroups –delete

gr –delete –g mygroup
: Deletes **mygroup**.

dgroups –intersect

dgroups –intersect –g 3 3.2
: Intersects thread 3.2 with group 3. If group 3 is a thread group, this command removes all threads except 3.2 from it; if it is a process group, this command removes all processes except process 3 from it.

f tW gr –i
: Intersects the focus thread with its workers group.

f gW gr –i –g mygroup
: Removes all nonworker threads from **mygroup**.

dgroups –list

dgroups –list
: Tells TotalView to display information about all named groups. For example:

```
ODD_P: {process 1 3}
EVEN_P: {process 2 4}
```

`gr –l *` Tells TotalView to display information about groups in the current focus.

```
1: {control 1 2 3 4}
2: {workers 1.1 1.2 1.3 1.4 2.1 2.2 2.3 2.4 3.1 \
                                3.2 3.3 3.4 4.1 4.2 4.3 4.4}
3: {share 1 2 3 4}
ODD_P: {process 1 3}
EVEN_P: {process 2 4}
```

dgroups –new

`gr –n t –g mygroup $GROUP($CGROUP(1))`
Creates a new thread group named **mygroup** containing all threads from all processes in the control group for process 1.

`set mygroup [dgroups –new]`
Creates a new process group that contains the current focus process.

dgroups –remove

`dgroups –remove –g 3 3.2`
Removes thread 3.2 from group 3.

`f W dgroups –add` Marks the current thread as being a worker thread.

`f W dgroups –r` Indicates that the current thread is not a workers thread.

dhalt

Suspends execution of processes

Format:

dhalt

Description:

The **dhalt** command stops all processes and threads in the current focus.

The command has no arguments.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
h	{dhalt}	Suspends execution
H	{dfocus g dhalt}	Group stop

Examples:

dhalt — Suspends execution of all *running* threads belonging to processes in the current focus. (Threads that are held at barriers are not affected.)

f t 1.1 h — Suspends execution of thread 1 in process 1. Note the difference between this command and **f 1.< dhalt**. If the focus is set as thread level, this command will halt the first user thread, which is probably thread 1.

dhold

Holds threads or processes

Format:

Holds processes

dhold –process

Holds threads

dhold –thread

Arguments:

–process	Indicates that processes in the current focus will be held. You can abbreviate **–process** to **–p**.
–thread	Indicates that threads in the current focus will be held. You can abbreviate **–thread** to **–t**.

Description:

The **dhold** command holds the threads and processes in the current focus.

NOTE You cannot hold system manager threads.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
hp	{dhold –process}	Holds the focus process.
HP	{f g dhold –process}	Holds all processes in the focus group.
ht	{f t dhold –thread}	Holds the focus thread.
HT	{f g dhold –thread}	Holds all threads in the focus group.
htp	{f p dhold –thread}	Holds all threads in the focus process.

Examples:

f W HT	Holds all worker threads in the focus group.
f s HP	Holds all processes in the share group.
f $mygroup/ HP	Holds all processes in the group identified by the contents of **mygroup**.

dkill

Terminates execution of processes

Format:

dkill

Description:

The **dkill** command terminates all processes in the current focus.

This command has no arguments.

Because the executables associated with the defined processes are still "loaded," typing the **drun** command restarts the processes.

The **dkill** command alters program state by terminating all processes in the affected set. In addition, TotalView destroys any spawned processes when the process that created them is killed. The **drun** command can only restart the initial process.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
k	{dkill}	Terminates a process's execution

Examples:

dkill — Terminates all threads belonging to processes in the current focus.

dfocus {p1 p3} dkill — Terminates all threads belonging to processes 1 and 3.

dlappend Appends lists elements to a TotalView variable

Format:

dlappend *variable-name value* [...]

Arguments:

variable-name	The variable to which values are being appended.
value	The values being appended.

Description:

The **dlappend** command appends list elements to a TotalView debugger variable. The **dlappend** command performs the same functions as the Tcl **lappend** command, differing in that **dlappend** will not create a new debugger variable. That is, the following Tcl command creates a variable named **foo**:

```
lappend foo 1 3 5
```

In contrast, the following command displays an error message:

```
dlappend foo 1 3 5
```

Examples:

dlappend TV::process_load_callbacks my_load_callback

Adds the **my_load_callback** function to the list of functions in the **process_load_callbacks** variable.

dlist

Displays source code lines

Format:

Displays code relative to the current list location

dlist [–n *num-lines*]

Displays code relative to a named place

dlist *source-loc* [–n *num-lines*]

Displays code relative to the current execution location

dlist –e [–n *num-lines*]

Arguments:

–n *num-lines*	Requests that this number of lines be displayed rather than the default value. (The default is the value of the **MAX_LIST** variable.) If *num-lines* is negative, lines before the current location are shown, and additional **dlist** commands will show preceding lines in the file rather than succeeding lines. For more information, see **dset** on page 200. This option also sets the value of the **MAX_LIST** variable to *num-lines*.
source-loc	Sets the location at which the CLI begins displaying information. This location is specified as a line number or as a string containing a file name, function name, and line number, each separated by **#** characters. For example: **file#func#line**. For more information, see "Qualifying Symbol Names" on page 80. Defaults are constructed if you omit parts of this specification.
–e	Sets the display location to include the current execution point of the thread of interest. If you used **dup** and **ddown** commands to select a buried stack frame, this location includes the PC (program counter) for that stack frame.

Description:

The **dlist** command displays lines relative to a place in the source code. (This position is called the *list location*.) The CLI prints this information; it is not returned. If neither *source-loc* nor **–e** is specified, the command contin-

ues where the previous list command left off. To display the thread's execution point, use **dlist –e**.

If you enter a file or procedure name, the listing begins at the file or procedure's first line.

The first time you use the **dlist** command after you focus on a different thread—or after the focus thread runs and stops again—the location changes to include the current execution point of the new focus thread.

Tabs in the source file are expanded as blanks in the output. The tab stop width is controlled by the **TAB_WIDTH** variable, which has a default value of 8. If **TAB_WIDTH** is set to –1, no tab processing is done, and tabs are displayed using their ASCII value.

All lines are shown with a line number and the source text for the line. The following symbols are also used:

@ An action point is set at this line.

> The PC for the current stack frame is at the indicated line and this is the leaf frame.

= The PC for the current stack frame is at the indicated line and this is a buried frame; this frame has called another function so that this frame is not the active frame.

These correspond to the marks shown in the backtrace displayed by **dwhere** that indicates the selected frame.

Here are some general rules:

- The initial display location is **main()**.
- The display location is set to the current execution location when the focus is on a different thread.

If the *source-loc* argument is not fully qualified, the CLI looks for it in the directories named in the CLI **EXECUTABLE_PATH** state variable.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
l	{dlist}	Displays lines

Examples:

These examples assume that **MAX_LIST** is at its initial value of 20.

dlist	Displays 20 lines of source code, beginning at the current list location. The list location is incremented by 20 when the command completes.
dlist 10	Displays 20 lines, starting with line 10 of the file corresponding to the current list location. Because an explicit value was used, the CLI ignores the previous command. The list location is changed to line 30.
dlist –n 10	Displays 10 lines, starting with the current list location. The value of the list location is incremented by 10.
dlist –n –50	Displays source code preceding the current list location; 50 lines are shown, ending with the current source code location. The list location is decremented by 50.
dlist do_it	Displays 20 lines in procedure **do_it**. The list location is changed so that it is the 20th line of the procedure.
dfocus 2.< dlist do_it	Displays 20 lines in the routine **do_it** associated with process 2. If the current source file were named **foo**, this could also be specified as **dlist foo#do_it**, naming the executable for process 2.
dlist –e	Displays 20 lines starting 10 lines above the current execution location.
f 1.2 l –e	Lists the lines around the current execution location of thread 2 in process 1.
dfocus 1.2 dlist –e –n 10	Produces essentially the same listing as the previous example, differing in that 10 lines are displayed.
dlist do_it.f#80 –n 10	Displays 10 lines, starting with line 80 in file **do_it.f**. The list location is updated to line 90.

dload

Loads debugging information

Format:

dload [–g *gid*] [–r *hname*] [–e] *executable*

Arguments:

–g *gid*	Sets the control group for the process being added to the group ID specified by *gid*. This group must already exist. (The CLI **GROUPS** variable contains a list of all groups.)
–r *hname*	The host on which the process will run. The CLI will launch a TotalView Debugger Server on the host machine if one is not already running there. See Chapter 5 of the TOTALVIEW USER GUIDE for information on the server launch commands.
–e	Tells the CLI that the next argument is a file name. You need to use this argument if the file name begins with a dash or only uses numeric characters.
executable	A fully or partially qualified file name for the file corresponding to the program.

Description:

The **dload** command creates a new TotalView process object for *executable*.

The **dload** command returns the TotalView ID for the new object.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
lo	{dload}	Loads debugging information

Examples:

dload do_this

Loads the debugging information for executable **do_this** into the CLI. After this command completes, the process does not yet exist and no address space or memory is allocated to it.

lo –g 3 –r other_computer do_this

Loads the debugging information for executable **do_this** executing on the **other_computer** machine into the CLI. This process is placed into group 3.

f g3 lo –r other_computer do_this
: Does not do what you would expect it to do because the **dload** command ignores the focus command.

dload –g $CGROUP(2) –r slowhost foo
: Loads another process based on image **foo** on machine **slowhost**. TotalView places this process in the same group as process 2.

dnext

Steps source lines, stepping over subroutines

Format:

dnext [*num-steps*]

Arguments:

num-steps — An integer number greater than 0, indicating the number of source lines to be executed.

The **dnext** command executes source lines; that is, it advances the program by steps (source line statements). However, if a statement in a source line invokes a routine, **dnext** executes the routine as if it were one statement; that is, it steps *over* the call.

The optional *num-steps* argument tells the CLI how many **dnext** operations it should perform. If you do not specify *num-steps*, the default is 1.

The **dnext** command iterates over the arenas in its focus set, performing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is **process** (**p**).

For more information on stepping in processes and threads, see **dstep on page 211**.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
n	{dnext}	Runs the thread of interest one statement while allowing other threads in the process to run.
N	{dfocus g dnext}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and steps one such "aligned" thread in each member one statement. The rest of the control group runs freely.

Alias	Definition	Meaning
nl	{dfocus L dnext}	Steps the process threads in "lockstep". This steps the thread of interest one statement and runs all threads in the process that are at the same PC as the thread of interest to the same statement. Other threads in the process run freely. The group of threads that are at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
NL	{dfocus gL dnext}	Steps "lockstep" threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one statement. Other threads in the control group run freely.
nw	{dfocus W dnext}	Steps worker threads in the process. This steps the thread of interest one statement, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
NW	{dfocus gW dnext}	Steps worker threads in the group. This steps the thread of interest one statement, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

dnext	Steps one source line.
n 10	Steps ten source line.
N	Steps one source line. It also runs all other processes in the group that are in the same lockstep group to the same line.

f t n	Steps the thread one statement.
dfocus 3. dnext	Steps process 3 one step.

dnexti

Steps machine instructions, stepping over subroutines

Format:

dnexti [*num-steps*]

Arguments:

num-steps	An integer number greater than 0, indicating the number of instructions to be executed.

Description:

The **dnexti** command executes machine-level instructions; that is, it advances the program by a single instruction. However, if the instruction invokes a subfunction, **dnexti** executes the subfunction as if it were one instruction; that is, it steps *over* the call. This command steps the thread of interest while allowing other threads in the process to run.

The optional *num-steps* argument tells the CLI how many **dnexti** operations it should perform. If you do not specify *num-steps*, the default is 1.

The **dnexti** command iterates over the arenas in the focus set, performing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is **process** (**p**).

For more information on stepping in processes and threads, see **dstep on page 211**.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
ni	{dnexti}	Runs the thread of interest one instruction while allowing other threads in the process to run.
NI	{dfocus g dnexti}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and steps one such "aligned" thread in each member one instruction. The rest of the control group runs freely.

Alias	Definition	Meaning
nil	{dfocus L dnexti}	Steps the process threads in "lockstep". This steps the thread of interest one instruction, and runs all threads in the process that are at the same PC as the thread of interest to the same statement. Other threads in the process run freely. The group of threads that are at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
NIL	{dfocus gL dnexti}	Steps "lockstep" threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one instruction. Other threads in the control group run freely.
niw	{dfocus W dnexti}	Steps worker threads in the process. This steps the thread of interest one instruction, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
NIW	{dfocus gW dnexti}	Steps worker threads in the group. This steps the thread of interest one instruction, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

dnexti	Steps one machine-level instruction.
ni 10	Steps ten machine-level instructions.
NI	Steps one instruction and runs all other processes in the group that were executing at that instruction to the next instruction as well.

f t n	Steps the thread one machine-level instruction.
dfocus 3. dnexti	Steps process 3 one machine-level instruction.

dout

Runs out from the current subroutine

Format:

dout [*frame-count*]

Arguments:

frame-count — Specifies that the thread returns out of this many levels of subroutine calls. If this number is omitted, the thread returns from the current level.

Description:

The **dout** command runs a thread until it returns:

- From the current subroutine.
- From one or more nested subroutines.

When process width is specified, TotalView allows all threads in the process that are not running to this goal to run free. Note that specifying process width is the default.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
ou	{dout}	Runs the thread of interest out of the current function while allowing other threads in the process to run.
OU	{dfocus g dout}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and runs one such "aligned" thread in each member out of the current function. The rest of the control group runs freely.

Alias	Definition	Meaning
oul	{dfocus L dout}	Runs the process threads in "lockstep". This runs the thread of interest out of the current function, and also runs all threads in the process that are at the same PC as the thread of interest out of the current function. Other threads in the process run freely. The group of threads that are at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
OUL	{dfocus gL dout}	Runs "lockstep" threads in the group. This runs all threads in the share group that are at the same PC as the thread of interest out of the current function. Other threads in the control group run freely.
ouw	{dfocus W dout}	Runs worker threads in the process. This runs the thread of interest out of the current function and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
OUW	{dfocus gW dout}	Runs worker threads in the group. This runs the thread of interest out of the current function and also runs all worker threads in the same share group out of the current function. All other threads in the control group run freely.

For additional information on the different kinds of stepping, see the **dstep** command information.

Examples:

f t ou	Runs the current thread of interest out of the current subroutine.
f p dout 3	Unwinds the process in the current focus out of the current subroutine to the routine three levels above it in the call stack.

dprint

Evaluates and displays information

Format:

Prints the value of a variable

dprint *variable*

Prints the value of an expression

dprint *expression*

Arguments:

variable — A variable whose value will be displayed. The variable can be local to the current stack frame or it can be global. If the variable being displayed is an array, you can qualify the variable's name with a slice that tells the CLI to display a portion of the array,

expression — A source-language expression to be evaluated and printed. Because *expression* must also conform to Tcl syntax, you must place it within quotes if it includes any blanks, and it must be enclosed in braces (**{}**) if it includes brackets (**[ ]**), dollar signs (**$**), quote characters (**"**), or any other Tcl special characters.

expression cannot contain calls to assembler, Fortran, C, or C++ functions.

Description:

The **dprint** command evaluates and displays a variable or an expression. The CLI interprets the expression by looking up the values associated with each symbol and applying the operators. The result of an expression can be a scalar value or an aggregate (array, array slice, or structure).

As the CLI displays data, it passes the data through a simple *more* process that prompts you after each screen of text is displayed. After a screen of data is displayed, you can press the Enter key to tell the CLI to continue displaying information. Entering **q** tells the CLI to stop printing this information.

Since the **dprint** command can generate a considerable amount of output, you may want to use the **capture** command described on page 128 to save the output into a variable.

Structure output appears with one field printed per line. For example:

```
sbfo = {
  f3 = 0x03 (3)
  f4 = 0x04 (4)
  f5 = 0x05 (5)
  f20 = 0x000014 (20)
  f32 = 0x00000020 (32)
}
```

Arrays are printed in a similar manner. For example:

```
foo = {
  [0][0] = 0x00000000 (0)
  [0][1] = 0x00000004 (4)
  [1][0] = 0x00000001 (1)
  [1][1] = 0x00000005 (5)
  [2][0] = 0x00000002 (2)
  [2][1] = 0x00000006 (6)
  [3][0] = 0x00000003 (3)
  [3][1] = 0x00000007 (7)
}
```

You can append a slice to the variable's name to tell the CLI that it should display a portion of an array. For example:

```
d1,<> p {master_array[::10]}
 master_array(::10) = {
  (1) = 1 (0x00000001)
  (11) = 1331 (0x00000533)
  (21) = 9261 (0x0000242d)
  (31) = 29791 (0x0000745f)
  (41) = 68921 (0x00010d39)
  (51) = 132651 (0x0002062b)
  (61) = 226981 (0x000376a5)
  (71) = 357911 (0x00057617)
  (81) = 531441 (0x00081bf1)
  (91) = 753571 (0x000b7fa3)
}
```

Note that the slice was placed within **{}** symbols. This prevents Tcl from trying to evaluate the information within the **[]** characters. You could, of course, escape the brackets; for example, **\[\]**.

The CLI evaluates the expression or variable in the context of each thread in the target focus. Thus, the overall format of **dprint** output is as follows:

```
first process/thread group:
    expression result

second process/thread group:
    expression result

...

last process/thread group:
    expression result
```

You can also use the **dprint** command to obtain values for your computer's registers. For example, on most architectures, **$r1** is register 1. You would obtain the contents of this registering by typing:

```
dprint \$r1
```

Notice that you must escape the **$** since the name of the register includes the **$**. This **$** is not the standard indicator that tells Tcl to fetch a variable's value. Appendix C, Architectures, in the TOTALVIEW USERS GUIDE lists the mnemonic names assigned to registers.

NOTE You do not need a $ when asking dprint to display your program's variables. For example, "$\$r1" looks for a Tcl variable named "$r1", not a program or TotalView variable named "$r1".

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
p	{dprint}	Evaluates and displays information.

Examples:

dprint scalar_y	Displays the values of variable **scalar_y** within all processes and threads in the current focus.
p argc	Displays the value of **argc**.
p argv	Displays the value of **argv**, along with the first string to which it points.

p {argv[argc-1]} Prints the value of **argv[argc-1]**. If the execution point is in **main()**, this is the last argument passed to **main()**.

dfocus p1 dprint scalar_y
Displays the values of variable **scalar_y** for the threads in process 1.

f 1.2 p arrayx Displays the values of the array **arrayx** for just the second thread in process 1.

for {set i 0} {$i < 100} {incr i} {p argv\[$i\]}
If **main()** is in the current scope, prints the program's arguments followed by the program's environment strings.

dptsets

Shows status of processes and threads in an array of P/T expressions

Format:

dptsets [*ptset_array*] ...

Arguments:

ptset_array	An optional array that indicates the P/T sets that will be shown. An element of the array can be a number or it can be a more complicated P/T expression. For more information, see "P/T Set Expressions" on page 63.

Description:

The **dptsets** command shows the status of each process and thread in a Tcl array of P/T expressions. These array elements are P/T expressions (see Chapter 4) and the elements' array indices are strings that label each element's section in the output. Using this array syntax is explored in the **Examples** section.

If you do not use the optional *ptset_array* argument, the CLI supplies a default array containing all P/T set designators. These designators are **error**, **existent**, **held**, **running**, **stopped**, **unheld**, and **watchpoint**.

Examples:

The following command displays information about processes and threads in the current focus:

```
d.1 <> dptsets
unheld:
1:      808694     Stopped     [fork_loopSGI]
  1.1:  808694.1   Stopped     PC=0x0d9cae64
  1.2:  808694.2   Stopped     PC=0x0d9cae64
  1.3:  808694.3   Stopped     PC=0x0d9cae64
  1.4:  808694.4   Stopped     PC=0x0d9cae64

existent:
1:      808694     Stopped     [fork_loopSGI]
  1.1:  808694.1   Stopped     PC=0x0d9cae64
  1.2:  808694.2   Stopped     PC=0x0d9cae64
  1.3:  808694.3   Stopped     PC=0x0d9cae64
  1.4:  808694.4   Stopped     PC=0x0d9cae64

watchpoint:
```

```
running:

held:

error:

stopped:
1:      808694      Stopped     [fork_loopSGI]
  1.1:  808694.1    Stopped     PC=0x0d9cae64
  1.2:  808694.2    Stopped     PC=0x0d9cae64
  1.3:  808694.3    Stopped     PC=0x0d9cae64
  1.4:  808694.4    Stopped     PC=0x0d9cae64
...
```

The following example creates a two-element P/T set array, then displays the results. Notice the labels in this example.

```
d1.<> set set_info(0) breakpoint(1)
breakpoint(1)
d1.<> set set_info(1) stopped(1)
stopped(1)
d1.<> dptsets set_info
0:
1:      892484    Breakpoint  [arraysSGI]
  1.1: 892484.1   Breakpoint  PC=0x10001544,
[arrays.F#81]

1:
1:      892484    Breakpoint  [arraysSGI]
  1.1: 892484.1   Breakpoint  PC=0x10001544,
[arrays.F#81]
```

The array index to **set_info** becomes a label identifying the kind of information being displayed. In contrast, the information within parentheses in the **breakpoint** and **stopped** functions identify the arena for which the function will return information.

Using numbers as array indices almost ensures that you will not remember what is being printed. The following almost identical example shows a better way to use these array indices.

```
d1.<> set set_info(my_breakpoints) breakpoint(1)
breakpoint(1)
d1.<> set set_info(my_stopped) stopped(1)
stopped(1)
d1.<> dptsets set_info
my_stopped:
1:      882547     Breakpoint  [arraysSGI]
  1.1: 882547.1  Breakpoint  PC=0x10001544,
[arrays.F#81]

my_breakpoints:
1:      882547     Breakpoint  [arraysSGI]
  1.1: 882547.1  Breakpoint  PC=0x10001544,
[arrays.F#81]
```

The following commands also create a two-element array. It differs in that the second element is the difference between three P/T sets.

```
d.1<> set mystat(system) a–gW
d.1<> set mystat(reallystopped) \
                          stopped(a)–breakpoint(a)–watchpoint(a)
d.1<> dptsets t mystat
system:
Threads in process 1 [regress/fork_loop]:
1.-1:    21587.[-1]      Running PC=0x3ff805c6998
1.-2:    21587.[-2]      Running PC=0x3ff805c669c
…
Threads in process 2 [regress/fork_loop.1]:
2.-1:    15224.[-1]      Stopped PC=0x3ff805c6998
2.-2:    15224.[-2]      Stopped PC=0x3ff805c669c
…

reallystopped:
2.2:     15224.2         Stopped PC=0x3ff800d5758
2.-1:    15224.[-1]      Stopped PC=0x3ff805c6998
2.-2:    15224.[-2]      Stopped PC=0x3ff805c669c
…
```

drerun

Restarts processes

Format:

drerun [*cmd_args*][*in_operation infile*]
[*out_operations outfile*]
[*error_operations errfile*]

Arguments:

cmd_args	The arguments to be used for restarting a process.
operations	The *in_operation*, *out_operations*, and *error_operations* are discussed in the *Description* section.
infile	If specified, indicates a file from which the launched processes will read information.
outfile	If specified, indicates the file into which the launched processes will write information.
errfile	If specified, indicates the file into which the launched processes will write error information.

Description:

The **drerun** command restarts the process that is in the current focus set from its beginning. The **drerun** command uses the arguments stored in the **ARGS** and **ARGS_DEFAULT** state variables. These are set every time the process is run with different arguments. Consequently, if you do not specify the arguments to be used when restarting the process, the CLI uses the arguments specified when the process was previously run. (See **drun** on page 197 for more information.)

The **dererun** command differs from the **drun** command in that

- If you do not specify an argument, **drerun** uses the default values. In contrast, the **drun** command clears the argument list for the program. This means that you cannot use an empty argument list with the **drerun** command to tell the CLI to restart a process and expect that no arguments will be used.
- If the process already exists, **drun** will not restart it. (If you must use the **drun** command, you must first kill the process.) In contrast, the **drerun** command will kill and then restart the process.

The arguments to this command are similar to the arguments used in the Bourne shell.

The *in_operation* is follows:

< *infile*	Reads from *infile* instead of **stdin**.

The *out_operations* are as follows:

> *outfile*	Sends output to *outfile* instead of **stdout**.
>& *outfile*	Sends output and error messages to *outfile* instead of **stdout** and **stderr**.
>>& *outfile*	Appends output and error messages to *outfile*.
>> *outfile*	Appends output to *outfile*.

The *error_operations* are:

2> *errfile*	Sends error messages to *errfile* instead of **stderr**.
2>>*errfile*	Appends error messages to *errfile*.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
rr	{drerun}	Restarts processes

Examples:

drerun Reruns the current process. Because arguments are not used, the process is restarted using its previous values.

rr –firstArg an_argument –aSecondArg a_second_argument
Reruns the current process. The default arguments are not used because replacement arguments are specified.

drestart

Restarts a checkpoint (SGI only)

Format:

drestart [*process-state*] [–no_unpark] [–g *gid*] [–r *host*]
[–ask_attach_parallel | –no_attach_parallel]
[–no_preserve_ids] *checkpoint-name*

Arguments:

process_state — Defines the state of the process both before and after the checkpoint. If you do not specify a process state, parallel processes are held immediately after the place where the checkpoint occurred. The CLI attaches to these created parallel processes. You can use one of the following options:

- –detach — While TotalView starts checkpointed process, it does not attach to them.
- –go — TotalView starts checkpointed parallel processes and attaches to them.
- –halt — TotalView stops checkpointed processes after it restarts them.

–no_unpark — Indicates that the checkpoint was created outside of TotalView or you that you used the **dcheckpoint** command's **–no_park** option when you created the checkpoint file.

–g *gid* — Names the control group into which TotalView places all created processes.

–r *host* — Names the remote host upon which the restart will occur.

–ask_attach_parallel — Asks if the CLI should automatically attach to the parallel processes being created. This is most often used in procedures.

–no_attach_parallel — Tells TotalView to attach only to the base process. That is, the CLI will not attach to the parallel processes being created.

–no_preserve_ids — Tells TotalView that it should use new IDs after it restarts the processes. If you omit this option, TotalView causes the process to use the same process, group, session, or **ash** IDs after restarting.

checkpoint-name — The name used when the checkpoint file was saved.

Description:

The **drestart** command restores and restarts all of the checkpointed processes. By default, the CLI will attach to the base process. Here are some of your choices.

- If there are parallel processes related to this base process, TotalView will attach to them.
- If you do not want the CLI to automatically attach to these parallel processes, use the **–no_attach_parallel** option.
- If you do not know if there are parallel processes or want the user to decide or if you are using this command within a Tcl procedure, you should use the **–ask_parallel_process** option.

Examples:

drestart check1 — Restarts the processes checkpointed in the **check1** file. The CLI automatically attaches to parallel processes.

drestart –no_unpark check1 — Restarts the processes checkpointed in the **check1** file. This file was either created outside of TotalView or it was created using the **–no_park** option.

drun

Starts or restarts processes

Format:

drun [*cmd_arguments*] [*in_operation infile*]
[*out_operations outfile*]
[*error_operations errfile*]

Arguments:

cmd_arguments	The argument list passed to the process.
operations	The *in_operation*, *out_operations*, and *error_operations* are discussed in the *Description* section.
infile	If specified, indicates a file from which the launched processes will read information.
outfile	If specified, indicates the file into which the launched processes will write information.
errfile	If specified, indicates the file into which the launched processes will write error information.

Description:

The **drun** command launches each process in the current focus and starts it running. The command arguments are passed to the processes, and I/O redirection for the program, if specified, will occur. Later in the session, you can use the **drerun** command to restart the program.

The arguments to this command are similar to the arguments used in the Bourne shell.

The *in_operation* is as follows:

< *infile*	Reads from *infile* instead of **stdin**.

The *out_operations* are as follows:

> *outfile*	Sends output to *outfile* instead of **stdout**.
>& *outfile*	Sends output and error messages to *outfile* instead of **stdout** and **stderr**.
>>& *outfile*	Appends output and error messages to *outfile*.
>> *outfile*	Appends output to *outfile*.

The *error_operations* are:

2> *errfile*	Sends error messages to *errfile* instead of **stderr**.
2>>*errfile*	Appends error messages to *errfile*.

In addition, the CLI uses the following state variables to hold the default argument list for each process.

ARGS_DEFAULT	The CLI sets this variable if you use the **–a** command-line option when you started the CLI or TotalView. (This option passes command-line arguments that TotalView will use when it invokes a process.) This variable holds the default arguments that TotalView passes to a process when the process has no default arguments of its own.
ARGS(*n*)	An array variable containing the command-line arguments. The index *n* is the process ID *n*. This variable holds a process's default arguments. It is always set by the **drun** command, and it is also contains any arguments you used when executing a **drerun** command.

If more than one process is launched with a single **drun** command, each receives the same command-line arguments.

In addition to setting these variables by using the **–a command-line** option or specifying *cmd_arguments* when you use this or the **drerun** command, you can modify these variables directly with the **dset** and **dunset** commands.

You can only use this command to tell TotalView that it should execute initial processes because TotalView cannot directly run processes that your program spawns. When you enter this command, initial process must be have terminated; if it was not terminated, you are told to kill it and retry. (You can, of course, use the **drerun** command.)

The first time you use the **drun** command, TotalView copies arguments to program variables. It also sets up any requested I/O redirection. If you reenter this command for processes that TotalVIew previously started—or issued for the first time for a process that was attached to using the **dattach** command—the CLI reinitializes your program.

Issues When Using IBM's poe

Both **poe** and the CLI can interfere with one another because each believes that it owns **stdin**. Because **poe** is trying to manage **stdin** on behalf of your processes, it continually reads from **stdin**, acquiring all characters that it sees. This means that the CLI will never see these characters. If your target process does not use **stdin**, you can use the **–stdinmode none** option. Unfortunately, this option is incompatible with **poe**'s **–cmdfile** option that is used when specifying **–pgmmodel mpmd**.

If you encounter these problems, you should redirect **stdin** within the CLI. For example:

```
drun < in.txt
```

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
r	{drun}	Starts or restarts processes.

Examples:

drun	Tells the CLI to begin executing processes represented in the current focus.
f {p2 p3} drun	Begins execution of processes 2 and 3.
f 4.2 r	Begins execution of process 4. Note that this is the same as **f 4 drun**.
dfocus a drun	Restarts execution of all processes known to the CLI. If they were not previously killed, you are told to use the **dkill** command and then try again.
drun < in.txt	Restarts execution of all processes in the current focus, setting them up to get standard input from file **in.txt**.

dset

Changes or views CLI state variables

Format:

Creates or changes a CLI state variable

dset [**–new**] *debugger-var value*

Views current CLI state variables

dset [*debugger-var*]

Arguments:

–new	Creates a variable if it does not already exist. If you omit this option and the variable does not exist, the CLI returns an error message.
debugger-var	Name of a CLI state variable.
value	Value to be assigned to *debugger-var.*

Description:

The **dset** command sets the value of CLI debugger variables.

If you type **dset** with no arguments, the CLI displays the names and current values for all TotalView CLI state variables. If you use only one argument, the CLI returns and displays the variable's value.

The second argument defines the value that will replace a variable's previous value. It must be enclosed in quotes if it contains more than one word.

If you do not use an argument, the CLI only displays variables in the current namespace. To show all variables in a namespace, just enter the namespace name immediately followed by a double colon; for example, **TV::**. You can also use an asterisk (*****) as a wildcard to indicate that the CLI should match more than one string; for example, **TV::g*** matches all variables beginning with **g** in the **TV** namespace.

For example, to view all variables in the **TV::** namespace, you would enter:

```
dset TV::
```

Similarly, you can view variables in a specific namespace by using the following command:

```
dset TV::GUI::
```

The rightmost double colons are required when obtaining listings for a namespace. If you omit them, Tcl assumes that you are requesting information on a variable. For example, **dset TV::GUI** looks for a variable named GUI in the **TV** namespace.

The state variables are:

ARGS(*dpid*) The argument strings that are passed to the process with TotalView ID **dpid** the next time it is launched.

ARGS_DEFAULT Contains the argument string passed to a new process that does not have an **ARGS(*dpid*)** variable defined.

BARRIER_STOP_ALL

Contains the default value for the **STOP_ALL** variable on newly created barrier points.

group: New barrier points will have the **stop_when_hit** flag set to **group**. When one thread reaches the barrier, TotalView stops all processes in its control group.

process: New barrier points just stop the process that hit the barrier.

none: New thread barrier points just stop the thread that hit the barrier.

BARRIER_STOP_WHEN_DONE

Contains the default value for a **–stop_when_done** option of commands that set barriers.

group: When a barrier is satisfied, TotalView stops all processes in the control group.

process: When a barrier is satisfied, TotalView stops the processes in the satisfaction set.

none: TotalView only stops the threads in the satisfaction set; other threads are not affected. For process barriers, there is no difference between **process** and **none**. In all cases, TotalView releases the satisfaction set when the barrier is satisfied.

CGROUP(*dpid*) Contains the control group for the process with the TotalView ID **dpid**. Setting this variable moves process **dpid** into a different control group. For example, the

following command moves process 3 into the same group as process 1:

```
dset CGROUP(3) $CGROUP(1)
```

COMMAND_EDITING

Enables some Emacs-like commands that you can use while editing text in the CLI. These editing commands are always available in the CLI window of GUI TotalView. However, they are only available within the stand-alone CLI if the terminal it is being run from supports cursor positioning and clear-to-end-of-line. The commands that you can use are:

^**A**: Moves to the beginning of the line.

^**B**: Moves one character backward.

^**D**: Deletes the character to the right of cursor.

^**E**: Moves to the end of the line.

^**F**: Moves one character forward.

^**K**: Deletes all text to the end of line.

^**N**: Retrieves the next entered command (only works after ^P).

^**P**: Retrieves the previously entered command.

^**R** or ^**L**: Redraws the line.

^**U**: Deletes all text from the cursor to the beginning of the line.

Rubout or **Backspace**: Deletes the character to the left of the cursor.

EXECUTABLE_PATH

Contains a colon-separated list containing the directories that TotalView searches when it looks for source and executable files.

GROUP(*gid*)

Contains a list containing the TotalView IDs for all members in group *gid*. User-created groups can be modified by directly using the **dgroups –add** and **dgroups –remove** commands.

The first element in the list indicates what kind of group it is, as follows:

control: The group of all processes in a program

lockstep: A group of threads that share the same PC

process: A user-created process group

share: The group of processes in one program that share the same executable image

thread: A user-created thread group

workers: The group of *worker* threads in a program

Elements that follow are either *pid*s (for process groups) or *pid*.*tid* pairs (for thread groups).

The *gid* is a simple number for most groups. In contrast, a lockstep group's ID number is of the form *pid*.*tid*. Thus, **GROUP(2.3)** contains the lockstep group for thread 3 in process 2. Note, however, that the CLI will not display lockstep groups when you use **dset** with no arguments—they are hidden variables.

The **GROUP(***id***)** variable is a read-write variable except when the group is not writable. Examples are lockstep, share, and control groups.

GROUPS — Contains a list that contains all TotalView groups IDs except for the lockstep groups.

LINES_PER_SCREEN

Defines the number of lines shown before the CLI stops printing information and displays its *more* prompt. The following values have special meaning:

0: No *more* processing occurs, and the printing does not stop when the screen fills with data.

NONE: This is a synonym for 0.

AUTO: The CLI uses the **tty** settings to determine the number of lines to display. This may not work in all cases. For example, Emacs sets the **tty** value to 0. If **AUTO** works improperly, you will need to explicitly set a value.

MAX_LIST	Defines the number of lines displayed in response to a **dlist** command.
PROMPT	Defines the CLI prompt. If you use brackets ([ ]) in the prompt, TotalView assumes the information within the brackets is a Tcl command and evaluates tit o obtain the prompt string.
PTSET	Defines the current focus.
SGROUP(*pid*)	Contains the group ID of the share group for process *pid*. TotalView decides which share group this is by looking at the control group for the process and the executable associated with it. You cannot directly modify this group.
SHARE_ACTION_POINT	Contains the default value for TotalView's internal **share_in_group** flag for newly created action points. If this value is *true*, an action point will be active across the group. If it is *false*, an action point is only active in the process upon which it is set. When a **dbarrier**, **dbreak**, or **dwatch** command is invoked using the default focus, this variable determines if the new action is shared.
STOP_ALL	Indicates the default value for the "stop all" attribute of newly created breakpoints and watchpoints, as follows: **group**: Stops the entire control group when the action point is hit. **process**: Stops the entire process when the action point is hit. **thread**: Only stops the thread that hit the action point. Note that **none** is a synonym for **thread**.
TAB_WIDTH	Indicates the number of spaces used to simulate a tab character when the CLI displays information.
THREADS(*pid*)	Contains a list of all threads in the process *pid*, in the form **{pid.1 pid.2 ...}**. This variable is read-only.

TOTALVIEW_ROOT_PATH
: Names the directory in which the TotalView executable is located.

TOTALVIEW_TCLLIB_PATH
: Contains a list containing the directories in which the CLI searches for TCL library components.

TOTALVIEW_VERSION
: Contains the version number and the type of computer architecture upon which TotalView is executing.

VERBOSE
: Controls the error message information displayed by the CLI. The values for this variable can be:

  INFO: Prints error, warnings, and informational messages. Informational message include data on dynamic libraries and symbols.

  WARNING: Only print errors and warnings.

  ERROR: Only print error messages.

  SILENT: Does not print error, warning, and informational messages. This also shuts off the printing of results from CLI commands. This should only be used when the CLI is run in batch mode.

WGROUP(*pid*)
: The group ID of the thread-group of worker threads associated with the process *pid*.

WGROUP(*pid.tid*)
: Either it contains the group ID of the worker sgroup in which thread *pid.tid* is a member or it contains 0 (zero), to indicate that thread *pid.tid* is not a worker thread. Storing a nonzero value in this variable marks a thread as a worker. In this case, the value that is read back will always be the ID of the workers group associated with the control group, regardless of the actual nonzero value assigned to it.

  Storing a zero value marks it as a nonworker.

The following table lists the default and permitted values for all CLI variables:

TABLE 8: **Defaults and Permitted Values for CLI Variables**

Debugger Variable	Permitted Values	Default
ARGS	A string	—
ARGS_DEFAULT	A string	—
BARRIER_STOP_ALL	**group**, **process**, or **thread**	group
BARRIER_STOP_WHEN_-DONE	**group**, **process**, or **thread**	group
CGROUP	A number	—
COMMAND_EDITING	true/false	false
EXECUTABLE_PATH	Any valid directory or directory path. To include the current setting, use **$EXECUTABLE_PATH**.	./:$PATH
GROUP	A Tcl array of lists indexed by the group ID. Each entry contains the members of one group.	—
GROUPS	A Tcl list of IDs. This is a read-only value and cannot be set.	—
LINES_PER_SCREEN	A positive integer, or the **AUTO** or **NONE** values.	AUTO
MAX_LIST	A positive integer	20
PROMPT	Any string. If you wish to access the value of **PTSET**, you must place the variable within brackets; that is, **[dset PTSET]**.	{[dfocus]> }
PTSET	This is a read-only value and cannot be set.	d1.<

TABLE 8: Defaults and Permitted Values for CLI Variables (cont.)

Debugger Variable	Permitted Values	Default
SGROUP	A number	—
SHARE_ACTION_POINT	True or false	true
STOP_ALL	**group**, **process**, or **thread**	**group**
TAB_WIDTH	A positive number. –1 indicates no tab expansion.	8
TOTALVIEW_ROOT_PATH	The location of the TotalView installation directory. This is a read-only variable and cannot be set.	
TOTALVIEW_TCLLIB_PATH	Any valid directory or directory path. To include the current setting, use **$TOTALVIEW_TCLLIB_PATH**.	The directory containing the CLI Tcl libraries
TOTALVIEW_VERSION	This is a read-only value and cannot be set.	
VERBOSE	**INFO**, **WARNING**, **ERROR**, and **SILENT**	**INFO**
WGROUP	A number	—

Examples:

dset PROMPT "Fixme% "
Sets the prompt to be **Fixme%** followed by a space.

dset *
Displays all CLI state variables and their current settings.

dset VERBOSE
Displays the current setting for output verbosity.

dset EXECUTABLE_PATH ../test_dir;$EXECUTABLE_PATH
Places **../test_dir** at the beginning of the previous value for the executable path.

dset TV::GUI::fixed_font_size 12
Sets the TotalView GUI so that it displays fixed fonts at 12 dpi. Commands such as this are often found in a startup file.

The following examples show the contents of a "*typical*" variable:

GROUP(1)	{ control 1 2 3 }	Control group. Members are processes 1, 2, and 3.
GROUP(2)	{ share 1 2 }	Share group. Members are processes 1 and 2.
GROUP(4)	{ thread 2.1 2.3 2.7}	General thread group. Members are threads 2.1, 2.3, and 2.7.
GROUP(5)	{ workers 17.3 17.4 17.12}	Thread workers group. Members are threads 17.3, 17.4, and 17.12.
GROUP(17)	{ process 2 5 9 }	A user-created group of processes. Members are processes 2, 5, and 9.

dstatus

Shows current status of processes and threads

Format:

dstatus

Description:

The **dstatus** command prints information about the current state of each process and thread in the current focus. The **ST** command is an alias for **dfocus g dstatus**, and acts as a group-status command.

If you have not changed the focus, the default width is *process*. In this case, **dstatus** shows the status for each thread in process 1. In contrast, if you set the focus to **g1.<**, the CLI displays the status for every thread in the control group containing process 1.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
st	{dstatus}	Shows current status
ST	{dfocus g dstatus}	Group status

Examples:

dstatus — Displays the status of all processes and threads in the current focus. For example:

```
1:      42898     Breakpoint  [arraysAIX]
  1.1:  42898.1     Breakpoint  \
          PC=0x100006a0, [./arrays.F#87]
```

f a st — Displays the status for all threads in all processes.

f p1 st — Displays the status of the threads associated with process 1. If the focus is at its default (**d1.<**), this is the same as typing **st**.

ST — Displays the status of all processes and threads in the control group containing the focus process. For example:

```
1:      773686     Stopped    [fork_loop_64]
  1.1:  773686.1     Stopped    PC=0x0d9cae64
  1.2:  773686.2     Stopped    PC=0x0d9cae64
  1.3:  773686.3     Stopped    PC=0x0d9cae64
  1.4:  773686.4     Stopped    PC=0x0d9cae64
```

```
  1.5:   773686.5     Stopped     PC=0x0d9cae64
2:       779490       Stopped     [fork_loop_64.1]
  2.1:   779490.1     Stopped     PC=0x0d9cae64
  2.2:   779490.2     Stopped     PC=0x0d9cae64
  2.3:   779490.3     Stopped     PC=0x0d9cae64
  2.4:   779490.4     Stopped     PC=0x0d9cae64
  2.5:   779490.5       Stopped     PC=0x0d9cae64
```

f W st — Shows status for all worker threads in the focus set. If the focus is set to **d1.<**, the CLI shows the status of each worker thread in process 1.

f W ST — Shows status for all worker threads in the control group associated with the current focus.

In this case, TotalView merges the **W** specifier with the **g** specifier in the **ST** alias. The results is the same as if you has entered **f gW st**.

f L ST — Shows status for every thread in the share group that is at the same PC as the thread of interest.

dstep

Steps lines, stepping into subfunctions

Format:

dstep [*num-steps*]

Arguments:

num-steps	An integer number greater than 0, indicating the number of source lines to be executed.

Description:

The **dstep** command executes source lines; that is, it advances the program by steps (source lines). If a statement in a source line invokes a subfunction, **dstep** steps into the function.

The optional *num-steps* argument tells the CLI how many **dstep** operations it should perform. If you do not specify *num-steps*, the default is 1.

The **dstep** command iterates over the arenas in the focus set by doing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is **process** (**p**).

If the width is **process**, the **dstep** command affects the entire process containing the thread to be stepped. Thus, while only one thread is stepped, all other threads contained in the same process also resume executing. In contrast, the **dfocus t dstep** command tells the CLI that it should just step the *thread of interest*.

NOTE On systems having identifiable manager threads, the "dfocus t dstep" command allows the manager threads to run as well as the thread of interest.

The action taken on each term in the focus list depends on whether its width is **thread**, **process**, or **group**, and on the group specified in the current focus. (If you do not explicitly specify a group, the default is the control group.)

If some thread hits an action point other than the goal breakpoint during a step operation, that ends the step.

thread width

Only the thread of interest is allowed to run. (This is not supported on all systems.)

process width (default)

The behavior depends on the group specified in the arena.

Process group	TotalView allows the entire process to run, and execution continues until the thread of interest arrives at its goal location. TotalView plants a temporary breakpoint at the goal location while this command executes. If another thread reaches this goal breakpoint first, your program continues to execute until the thread of interest reaches the goal.
Thread group	TotalView runs all threads in the process that are in that group to the same goal as the thread of interest. If a thread arrives at the goal that is not in the group of interest, it also stops there. The group of interest specifies the set of threads for which TotalView will wait. This means that the command does not complete until all threads in the group of interest are at the goal.

group width

The behavior depends on the group specified in the arena.

Process group	TotalView examines that group, and each process having a thread stopped at the same location as the thread of interest is identified. One matching thread from each matching process is selected. TotalView then runs all processes in the group, and waits until the thread of interest arrives at its goal location, and each selected thread also arrives there.
Thread group	The behavior is similar to process width behavior except that all processes in the program control group will run, rather than just the process of interest. Regardless of what threads are in the group of interest, TotalView only waits for threads that are in the same share group as the thread of interest. This is because it is not useful to run threads executing in different images to the same goal.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
s	{dstep}	Runs the thread of interest one statement while allowing other threads in the process to run.
S	{dfocus g dstep}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and steps one such "aligned" thread in each member one statement. The rest of the control group runs freely.
sl	{dfocus L dstep}	Steps the process threads in "lockstep". This steps the thread of interest one statement, and runs all threads in the process that are at the same PC as the thread of interest to the same statement. Other threads in the process run freely. The group of threads that are at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
SL	{dfocus gL dstep}	Steps "lockstep" threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one statement. Other threads in the control group run freely.
sw	{dfocus W dstep}	Steps worker threads in the process. This steps the thread of interest one statement, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.

Alias	Definition	Meaning
SW	{dfocus gW dstep}	Steps worker threads in the group. This steps the thread of interest one statement, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

dstep	Executes the next source line, stepping into any procedure call that is encountered. While only the current thread is stepped, other threads in the process run.
s 15	Executes the next 15 source lines.
f p1.2 dstep	Steps thread 2 in process 1 by one source line. This also resumes execution of all other threads in process 1; they are halted as soon as thread 2 in process 1 executes its statement.
f t1.2 s	Steps thread 2 in process 1 by one source line. No other threads in process 1 execute.

dstepi

Steps machine instructions, stepping into subfunctions

Format:

dstepi [*num-steps*]

Arguments:

num-steps — An integer number greater than 0, indicating the number of instructions to be executed.

Description:

The **dstepi** command executes assembler instruction lines; that is, it advances the program by single instructions.

The optional *num-steps* argument tells the CLI how many **dstepi** operations it should perform. If you do not specify *num-steps*, the default is **1**.

For more information, see **dstep on page 211**.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
si	{dstepi}	Runs the thread of interest one instruction while allowing other threads in the process to run.
SI	{dfocus g dstepi}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and steps one such "aligned" thread in each member one instruction. The rest of the control group runs freely.
sil	{dfocus L dstepi}	Steps the process threads in "lockstep". This steps the thread of interest one instruction, and runs all threads in the process that are at the same PC as the thread of interest to the same instruction. Other threads in the process run freely. The group of threads that are at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.

Alias	Definition	Meaning
SIL	{dfocus gL dstepi}	Steps "lockstep" threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one instruction. Other threads in the control group run freely.
siw	{dfocus W dstepi}	Steps worker threads in the process. This steps the thread of interest one instruction, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
SIW	{dfocus gW dstepi}	Steps worker threads in the group. This steps the thread of interest one instruction, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

dstepi	Executes the next machine instruction, stepping into any procedure call that is encountered. While only the current thread is stepped, other threads in the process are allowed to run.
si 15	Executes the next 15 instructions.
f p1.2 dstepi	Steps thread 2 in process 1 by one instruction. This also resumes execution of all other threads in process 1; they are halted as soon as thread 2 in process 1 executes its instruction.
f t1.2 si	Steps thread 2 in process 1 by one instruction. No other threads in process 1 execute.

dunhold

Releases a held process or thread

Format:

Releases a process

dunhold –process

Releases a thread

dunhold –thread

Arguments:

–process	Indicates that TotalView should release processes in the current focus. **–process** can be abbreviated to **–p**.
–thread	Indicates that TotalView should release threads in the current focus. **–thread** can be abbreviated to **–t**.

Description:

The **dunhold** command releases the threads or processes in the current focus. Note that system manager threads cannot be held or released.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
uhp	{dfocus p dunhold -process}	Releases the process of interest.
UHP	{dfocus g dunhold -process}	Releases the processes in the focus group.
uht	{dfocus t dunhold –thread}	Releases the thread of interest.
UHT	{dfocus g dunhold –thread}	Releases all threads in the focus group.
uhtp	{dfocus p dunhold –thread}	Releases the threads in the current process.

Examples:

f w uhtp	Unholds all worker threads in the focus process.
htp; uht	Holds all threads in the focus process except the thread of interest.

dunset

Restores default settings for state variables

Format:

Restores a CLI variable to its default value

dunset *debugger-var*

Restores all CLI variables to their default values

dunset –all

Arguments:

debugger-var	Name of the CLI state variable whose default setting is being restored.
–all	Restores the default settings of the CLI state variables.

Description:

The **dunset** command reverses the effects of any previous **dset** commands, restoring CLI state variables to their default settings.

NOTE Because user-defined state variables have no default values, the CLI deletes them.

Tcl variables (those created with the Tcl **set** command) are, of course, unaffected by this command.

If you use the **–all** option, the **dunset** command affects all changed CLI state variables, restoring them to the settings that existed when the CLI session began. Similarly, specifying *debugger-var* tells the CLI to restore that one variable.

Examples:

dunset PROMPT	Restores the prompt string to its default setting; that is, **{[dfocus]>}**.
dunset –all	Restores all CLI state variables to their default settings.

duntil

Runs the process until a target place is reached

Format:

Runs to a line

duntil *line-number*

Runs to an address

duntil –address *addr*

Runs into a function

duntil *proc-name*

Arguments:

line-number	A line number in your program.
–address *addr*	An address in your program.
proc-name	The name of a procedure, function, or subroutine in your program.

Description:

The **duntil** command runs the thread of interest until execution reaches a line or absolute address, or until it enters a function.

If you use a process or group width, all threads in the process or group that are not running to the goal are allowed to run. If one of the "secondary" threads arrives at the goal before the thread of interest, it continues running, ignoring this goal. In contrast, if you specify thread width, only the thread of interest runs.

The **duntil** command differs from other step commands when you apply it to a group.

Process group	TotalView runs the entire group and the CLI waits until all processes in the group have at least one thread that has arrived at the goal breakpoint. This lets you *sync up* all the processes in a group in preparation for group-stepping them.
Thread group	TotalView runs the process (for **p** width) or the control group (for **g** width) and waits until all the running threads in the group of interest arrive at the goal.

The differences between this command and other stepping commands are:

- **Process Group Operation**: TotalView examines the thread of interest to see if it is already at the goal. If it is, TotalView does not run the process of interest. Similarly, TotalView examines all other processes in the share group, and it only run processes that do not have a thread at the goal. It also runs members of the control group that are not in the share group.
- **Group-Width Thread Group Operation**: TotalView identifies all threads in the entire control group that are not at the goal. Only those threads are run. While TotalView runs share group members in which all worker threads are already at the goal, it does not run the workers. TotalView also runs processes in the control group that are outside the share group. The **duntil** command operation ends when all members of the focus thread group are at the goal.
- **Process-Width Thread Group Operation**: TotalView identifies all threads in the entire focus process that are not already at the goal. Only those threads run. The **duntil** command operation ends when all threads in the process that are also members of the focus group arrive at the goal.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
un	{duntil}	Runs the thread of interest until it reaches a target while allowing other threads in the process to run.
UN	{dfocus g duntil}	Runs the entire control group until every process in the share group has at least one thread at the goal. Processes have a thread at the goal do not run.

Alias	Definition	Meaning
unl	{dfocus L duntil}	Runs the thread of interest until it reaches the target, and runs all threads in the process that are at the same PC as the thread of interest to the same target. Other threads in the process run freely. The group of threads that are at the same PC is called the *lockstep group*. This does not force process width. If the default focus is set to **group**, this runs the group.
UNL	{dfocus gL duntil}	Runs "lockstep" threads in the share group until they reach the target. Other threads in the control group are allowed to run freely.
unw	{dfocus W duntil}	Runs worker threads in the process to a target. The nonworker threads in the process run freely. This does not force process width. If the default focus is set to **group**, this runs the group.
UNW	{dfocus gW duntil}	Runs worker threads in the same share group to a target. All other threads in the control group run freely.

Examples:

UNW 580	Lines up all worker threads at line 580.
un buggy_subr	Runs to the start of the **buggy_subr** routine.

dup

Moves up the call stack

Format:

dup [*num-levels*]

Arguments:

num-levels	Number of levels to move up. The default is **1**.

Description:

The **dup** command moves the current stack frame up one or more levels. It also prints the new frame number and function.

Call stack movements are all relative, so **dup** effectively "moves up" in the call stack. ("Up" is in the direction of **main()**.)

Frame 0 is the most recent—that is, currently executing—frame in the call stack, frame 1 corresponds to the procedure that invoked the currently executing one, and so on. The call stack's depth is increased by one each time a procedure is entered, and decreased by one when it is exited. The effect of **dup** is to change the context of commands that follow. For example, moving up one level lets you access variables that are local to the procedure that called the current routine.

Each **dup** command updates the frame location by adding the appropriate number of levels.

The **dup** command also modifies the current list location to be the current execution location for the new frame, so a subsequent **dlist** displays the code surrounding this location. Entering **dup 2** (while in frame 0) followed by a **dlist**, for instance, displays source lines centered around the location from which the current routine's parent was invoked. These lines will be in frame 2.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
u	{dup}	Moves up the call stack

Examples:

dup

Moves up one level in the call stack. As a result, subsequent **dlist** commands refer to the procedure that invoked this one. After this command executes, it displays information about the new frame. For example:

```
1 check_fortran_arrays_ PC=0x10001254,
    FP=0x7fff2ed0 [arrays.F#48]
```

dfocus p1 u 5

Moves up five levels in the call stack for each thread involved in process 1. If fewer than five levels exist, the CLI moves up as far as it can.

dwait

Blocks command input until the target processes stop

Format:

dwait

Description:

The **dwait** command tells the CLI to wait for all threads in the current focus to stop or exit. Generally, this command treats the focus identically to other CLI commands.

If you interrupt this command—typically by typing Ctrl-C—the CLI manually stops all processes in the current focus before it returns.

Unlike most other CLI commands, this command blocks additional CLI input until the blocking action is complete.

Examples:

dwait — Blocks further command input until all processes in the current focus have stopped (that is, none of their threads are still **running**).

dfocus {p1 p2} dwait — Blocks command input until processes 1 and 2 stop.

dwatch

Defines a watchpoint

Format:

Defines a watchpoint for a variable

dwatch *variable* [**–length** *byte-count*] [**–g** | **–p** | **–t**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Defines a watchpoint for an address

dwatch –address *addr* **–length** *byte-count* [**–g** | **–p** | **–t**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Arguments:

variable	A symbol name corresponding to a scalar or aggregate identifier, an element of an aggregate, or a dereferenced pointer.
–address *addr*	An absolute address in the file.
–length *byte-count*	The number of bytes to watch. If a variable is named, the default is the variable's byte length. If you are watching a variable, you only need to specify the amount of storage to watch if you want to override the default value.
–g	Tells TotalView to stop all processes in the process's control group when the watchpoint is hit.
–p	Tells TotalView to stop the process that hit this watchpoint.
–t	Tells TotalView to stop the thread that hit this watchpoint.
–l *lang*	Specifies the language used when writing an expression. The values you can use for *lang* are **c**, **c++**, **f77**, **f90**, and **asm**, for C, C++, FORTRAN 77, Fortran-9x, and assembler, respectively. If you do not use a language code, TotalView picks one based on the variable's type. If only an address is used, TotalView uses the C language. Not all languages are supported on all systems.
–e *expr*	When the watchpoint is triggered, evaluates *expr* in the context of the thread that hit the watchpoint.

In most cases, you need to enclose the expression in braces ({ }).

–t *type* The data type of **$oldval**/**$newval** in the expression.

Description:

A **dwatch** command defines a watchpoint on a memory location where the specified variables are stored. The watchpoint triggers whenever the value of the variables changes. The CLI returns the ID of the newly created watchpoint.

NOTE Watchpoints are not available on Alpha Linux and HP.

The default action is controlled by the **STOP_ALL** variable.

The watched variable can be a scalar, array, record, or structure object, or a reference to a particular element in an array, record, or structure. It can also be a dereferenced pointer variable.

The CLI lets you obtain a variable's address in the following two ways if your application demands that you specify a watchpoint with an address instead of a variable name:

- **dprint** *&variable*
- **dwhat** *variable*

The **dprint** command displays an error message if the variable is within a register.

NOTE Chapter 8 of the TOTALVIEW USERS GUIDE contains additional information on watchpoints.

If you do not use the **–length** modifier, the CLI uses the length attribute from the program's symbol table. This means that the watchpoint applies to the data object named; that is, specifying the name of an array lets you watch all elements of the array. Alternatively, you can specify that a certain number of bytes be watched, starting at the named location.

NOTE In all cases, the CLI watches addresses. If you specify a variable as the target of a watchpoint, the CLI resolves the variable to an absolute address. If you are watching a local stack variable, the position being watched is just where the variable happened to be when space for the variable was allocated.

The focus establishes the processes (not individual threads) for which the watchpoint is in effect.

The CLI prints a message showing the action point identifier, the location being watched, the current execution location of the triggering thread, and the identifier of the triggering threads.

One possibly confusing aspect of using expressions is that their syntax differs from that of Tcl. This is because you will need to embed code written in Fortran, C, or assembler within Tcl commands. In addition, your expressions will often include TotalView intrinsic functions.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
wa	{dwatch}	Defines a watchpoint.

Examples:

For these examples, assume that the current process set at the time of the **dwatch** command consists only of process 2, and that **p** is a global variable that is a pointer.

dwatch *p

Watches the address stored in pointer **p** at the time the watchpoint is defined, for changes made by process 2. Only process 2 is stopped. Note that the watchpoint location does not change when the value of **p** changes.

dwatch {*p}

Performs the same action as the previous example. Because the argument to **dwatch** contains a space, Tcl requires that you place the argument within braces.

dfocus {p2 p3} wa *p

Watches the address pointed to by **p** in processes 2 and 3. Because this example does not contain either a **–p** or **–g** option, the value of the **STOP_ALL** state variable lets the CLI know if it should stop processes or groups.

dfocus {p2 p3 p4} dwatch –p *p

Watches the address pointed to by **p** in processes 2, 3, and 4. The **–p** option indicates that only the process triggering the watchpoint is stopped.

wa * aString –length 30 –e {goto $447}

Watches 30 bytes of data beginning at the location pointed to by **aString**. If any of these bytes change, execution control transfers to line 447.

wa my_vbl –type long –e { if ($newval == 0x11ffff38) $stop; }

Watches the **my_vbl** variable and triggers when **0x11ffff38** is stored into it.

wa my_vbl –e { if (my_vbl == 0x11ffff38) $stop; }

Performs the same function as the previous example. Note that this tests the variable directly rather than by using **$newval**.

dwhat

Determines what a name refers to

Format:

dwhat *symbol-name*

Arguments:

symbol-name	Fully or partially qualified name specifying a variable, procedure, or other source code symbol.

Description:

The **dwhat** command tells the CLI to display information about a named entity in a program. The displayed information contains the name of the entity and a description of the name. The examples that follow show many of the kinds of elements that this command can display.

NOTE To view information on CLI state variables or aliases, you need to use the dset or alias commands.

The focus constrains the query to a particular context.

The default width for this command is **thread** (**t**).

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
wh	{dwhat}	Determines what a name refers to.

Examples:

These examples show what the CLI displays when you enter one of the indicated commands.

dprint timeout

```
timeout = {
    tv_sec = 0xc0089540 (-1073179328)
    tv_usec = 0x000003ff (1023)
}
```

dwhat timeout

In thread 1.1:

```
Name: timeout; Type: struct timeval; Size: 8
bytes; Addr: 0x11fffefc0
    Scope: #fork_loop.cxx#snore \
        (Scope class: Any)
    Address class: auto_var (Local variable)
```

```
wh timeval          In process 1:

                    Type name: struct timeval; Size: 8 bytes; \
                            Category: Structure
                        Fields in type:
                        {
                        tv_sec      time_t      (32 bits)
                        tv_usec     int         (32 bits)
                        }
dlist               20      float field3_float;
                    21      double field3_double;
                    22  en_check en1;
                    23
                    24  };
                    25
                    26  main ()
                    27  {
                    28      en_check vbl;
                    29      check_struct s_vbl;
                    30  >   vbl = big;
                    31      s_vbl.field2_char = 3;
                    32      return (vbl + s_vbl.field2_char);
                    33  }
p vbl               vbl = big (0)
wh vbl              In thread 2.3:

                    Name: vbl; Type: enum en_check; \
                            Size: 4 bytes; Addr: Register 01
                        Scope: #check_structs.cxx#main \
                            (Scope class: Any)
                        Address class: register_var (Register variable)
wh en_check         In process 2:

                    Type name: enum en_check; Size: 4 bytes; \
                            Category: Enumeration
                        Enumerated values:
                            big         = 0
                            little      = 1
                            fat         = 2
                            thin        = 3
```

p s_vbl

```
s_vbl = {
    field1_int = 0x800164dc (-2147392292)
    field2_char = '\377' (0xff, or -1)
    field2_chars = "\003"
    <padding> = '\000' (0x00, or 0)
    field3_int = 0xc0006140 (-1073716928)
    field2_uchar = '\377' (0xff, or 255)
    <padding> = '\003' (0x03, or 3)
    <padding> = '\000' (0x00, or 0)
    <padding> = '\000' (0x00, or 0)


    field_sub = {
        field1_int = 0xc0002980 (-1073731200)
        <padding> = '\377' (0xff, or -1)
        <padding> = '\003' (0x03, or 3)
        <padding> = '\000' (0x00, or 0)
        <padding> = '\000' (0x00, or 0)
        field2_long = 0x0000000000000000 (0)
    ...
}
```

wh s_vbl

In thread 2.3:

```
Name: s_vbl; Type: struct check_struct; Size: 80 \
        bytes; Addr: 0x11ffff240
    Scope: #check_structs.cxx#main \
        (Scope class: Any)
    Address class: auto_var (Local variable)
```

wh check_struct

In process 2:

```
Type name: struct check_struct; Size: 80 bytes; \
        Category: Structure
    Fields in type:
    {
    field1_int          int             (32 bits)
    field2_char         char            (8 bits)
    field2_chars        <string>[2]     (16 bits)
    <padding>           <char>          (8 bits)
    field3_int          int             (32 bits)
    field2_uchar        unsigned char   (8 bits)
    <padding>           <char>[3]       (24 bits)
    field_sub           struct sub_struct (320 bits){
```

```
            field1_int     int              (32 bits)
            <padding>   <char>[4]    (32 bits)
            field2_long    long             (64 bits)
            field2_ulong   unsigned long (64 bits)
            field3_uint    unsigned int   (32 bits)
            en1            enum en_check (32 bits)
            field3_double double          (64 bits)
        }
        …
}
```

dwhere

Displays locations in the call stack

Format:

dwhere [*num-levels*] [–args]

Arguments:

num-levels	Restricts output to this number of levels of the call stack.
–args	Displays argument names and values in addition to program location information. If you omit this option, arguments are not shown.

Description:

The **dwhere** command prints the current execution locations and the call stacks—or sequences of procedure calls—which led to that point. Information is shown for threads in the current focus, with the default being to show information at the thread level.

Arguments control the amount of command output in two ways:

- The *num-levels* argument lets you control how many levels of the call stacks are displayed, counting from the uppermost (most recent) level.
- The **–args** option tells the CLI that it should also display procedure argument names and values for each stack level.

A **dwhere** command with no arguments or options displays the call stacks for all threads in the target set.

Output is generated for each thread in the target focus.

Command alias:

You may find the following alias useful:

Alias	Definition	Meaning
w	{dwhere}	Displays the current location

Examples:

dwhere
Displays the call stacks for all threads in the current focus.

dfocus 2.1 dwhere 1
Displays just the most recent level of the call stack corresponding to thread 1 in process 2. This shows

	just the immediate execution location of a thread or threads.
w 1 –args	Displays the current execution locations (one level only) of threads in the current focus together with the names and values of any arguments that were passed into the current procedures.
f p1.< w 5	Displays the most recent five levels of the call stacks for all threads involved in process 1. If the depth of any call stack is less than five levels, all of its levels are shown. This command is a slightly more complicated way of saying **f p1 w 5** because specifying a process width tells **dwhere** to ignore the thread indicator.

dworker

Adds or removes a thread from a workers group

Format:

dworker { *number* | *boolean* }

Arguments:

number	If positive, marks the thread of interest as a worker thread by inserting it into the workers group.
boolean	If **true**, marks the thread of interest as a worker thread by inserting it into the workers group. If **false**, mark the thread as being a nonworker thread by removing it from the workers group.

Description:

The **dworker** command inserts or removes a thread from the workers group.

If *number* is **0** or **false**, this command marks the thread of interest as a non-worker thread by removing it from the workers group. If *number* is **true** or is a positive value, this command marks the thread of interest as a worker thread by inserting it in the workers group.

Note that moving a thread into or out of the workers group has no effect on whether the thread is a "manager" thread. Manager threads are threads that are created by the pthreads package to manage other threads; they never execute user code, and cannot normally be controlled individually. TotalView automatically inserts all threads that are not manager threads into the workers group.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
wof	{dworker false}	Removes the focus thread from the workers group.
wot	{dworker true}	Inserts the focus thread into the workers group.

errorCodes

Returns or raises TotalView error information

Format:

Returns a list of all error code tags

TV::errorCodes

Returns or raises error information

TV::errorCodes *number_or_tag* [**–raise** [*message*]]

Arguments:

number_or_tag	Enters an error code mnemonic tag or its numeric value.
–raise	Raises the corresponding error. If you append a *message*, TotalView returns this string. Otherwise, TotalView uses the human-readable string for the error.
message	An optional string used when raising an error.

Description:

The **TV::errorCodes** command lets you manipulate the TotalView error code information placed in the Tcl **errorCodes** variable. The CLI sets this variable after every command error. Its value is intended to be easy to parse in a Tcl script.

When the CLI or TotalView returns an error, **errorCode** is set to a list whose format is:

TOTALVIEW *error-code subcodes... string*

The contents of this lists are as follows:

- The first list element is always **TOTALVIEW**.
- The second is always the error code.
- *subcodes* are not used at this time.
- The last element is a string describing the error.

With a tag or number, this command returns a list containing the mnemonic tag, the numeric value of the tag, and the string associated with the error.

The **–raise** option tells the CLI to raise an error. If you add a message, that message is used as the return value; otherwise, the CLI uses its textual

explanation for the error code. This provides an easy way to return TotalView-style errors from a script.

Examples:

```
foreach e [TV::errorCodes] {
        puts [eval format {"%20s %2d %s"} [TV::errorCodes $e]] }
```

Displays a list of all TotalView error codes.

exit

Terminates the debugging session

Format:

exit [–force]

Arguments:

–force	Tells the CLI that TotalView should exit without asking permission.

Description:

The **exit** command terminates the TotalView session.

After executing this command, the CLI asks if it is all right to exit. If you answer yes, TotalView exits. If you had entered the CLI from the TotalView GUI, this command also closes the GUI window.

NOTE Type Ctrl-D to exit from the CLI window without exiting from TotalView.

Any processes and threads that were created by the CLI are destroyed. Any processes that existed prior to the debugging session (that is, were attached by the CLI as part of a **dattach** operation) are detached and left executing.

The **exit** and **quit** commands are interchangeable; they both do exactly the same thing.

Examples:

exit	Exits from the CLI, leaving any "attached" processes running.

focus_groups

Returns a list of groups in the current focus

Format:

TV::focus_groups

Description:

The **TV::focus_groups** command returns a list of all groups in the current focus.

Examples:

f d1.< TV::focus_groups

Returns a list containing one entry, which will be the ID of the control group for process 1.

focus_processes

Returns a list of processes in the current focus

Format:

TV::focus_processes [–all | –group | –process | –thread]

Arguments:

–all	Changes the default width to **all**.
–group	Changes the default width to **group**.
–process	Changes the default width to **process**.
–thread	Changes the default width to **thread**.

Description:

The **TV::focus_processes** command returns a list of all processes in the current focus. If the focus width is something other than **d** (default), the focus width determines the set of processes returned. If the focus width is **d**, the **TV::focus_processes** command returns process width. Using any of the options changes the default width.

Examples:

f g1.< TV::focus_processes

Returns a list containing all processes in the same control as process 1.

focus_threads

Returns a list of threads in the current focus

Format:

TV::focus_threads [–all | –group | –process | –thread]

Arguments:

–all	Changes the default width to **all**.
–group	Changes the default width to **group**.
–process	Changes the default width to **process**.
–thread	Changes the default width to **thread**.

Description:

The **TV::focus_threads** command returns a list of all threads in the current focus. If the focus width is something other than **d** (default), the focus width determines the set of threads returned. If the focus width is **d**, **TV::focus_threads** returns thread width. Using any of the options changes the default width.

Examples:

f p1.< TV::focus_threads

Returns a list containing all threads in process 1.

group

Sets and gets group properties

Format:

TV::group *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Lists the subcommands that you can use. The CLI responds by displaying the four subcommands shown here. Do not use additional arguments with this subcommand.
- **get** — Gets the values of one or more group properties. The *other-args* argument can include one or more property names. The CLI returns these values for these properties in a list in the same order as you entered the property names.

  If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each group.
- **properties** — Lists the properties that the CLI can access. Do not use additional arguments with this option.
- **set** — Sets the values of one or more properties. The *other-args* subcommand is a sequence of property name and value pairs.

object-id — The group ID. If you use the **–all** option, the operation is carried out on all groups in the current focus.

other-args — Arguments required by the **get** and **set** subcommands.

Description:

The **TV::group** command lets you examine and set group properties and states. These states and properties are:

- **count** — The number of members in a group.
- **id** — The ID of the object.
- **member_type** — The type of the group's members, either **process** or **thread**.

member_type_values
: Returns a list of all possible values for the **member_type** property.

members
: A list of a group's processes or threads.

type
: The group's type. Possible values are **control**, **lockstep**, **share**, **user**, and **workers**.

type_values
: Returns a list of all possible values for the **type** property.

Examples:

TV::group get 1 count
: Returns the number of objects in group 1.

help

Displays help information

Format:

help [*topic*]

Arguments:

topic	The topic or command for which the CLI prints information.

Description:

The **help** command prints information about the specified topic or command. If you do no use an argument, the CLI displays a list of the topics for which help is available.

If more than one screen of data would be displayed, the CLI fills the screen with data and then displays a *more* prompt. You can then press Enter to see more data or enter **q** to return to the CLI prompt.

When you type a topic name, the CLI will attempt to complete what you type. **help** also allows you to enter one of the CLI's built-in aliases. For example:

```
d1.<> he a
Ambiguous help topic "a". Possible matches:
    alias accessors arguments addressing_expressions
d1.<> he ac
"ac" has been aliased to "dactions":
dactions [ bp-ids ... ] [ -at <source-loc> ] [ -disabled | -enabled
]
    Default alias: ac
...
d1.<> he acc
    The following commands provide access to the properties
of
    TotalView objects:
    ...
```

You can use the **capture** command to place help information into a variable.

Command alias:

You may find the following aliases useful:

Alias	Definition	Meaning
he	{help}	Displays help information.

Examples:

help help Prints information describing the **help** command.

hex2dec Converts to decimal

Format:

TV::hex2dec *number*

Arguments:

number	A hexadecimal number.

Description:

Converts a hexadecimal number into decimal. You can type **0x** before this value. The CLI correctly manipulates 64-bit values, regardless of the size of a **long**.

image

Sets and gets image properties

Format:

TV::image *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

add — Adds an object to an image. The *object-id* argument is required; *other-args* is followed by two arguments. The first is the object class of the image. At this release, this type can only be **prototype**. The second is the prototype's ID. (This is illustrated in the E*xamples* section.)

commands — Lists the subcommands that you can use. The CLI responds by displaying the six subcommands shown here. Do not use additional arguments with this subcommand.

get — Gets the values of one or more image properties. The *other-args* argument can include one or more property names. The CLI returns these values for these properties in a list in the same order as you entered the property names.

If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.

lookup — Looks up an object in the image and returns a list of IDs of matching objects. The *object-id* argument is required; *other-args* contains two arguments. The first is the object class of the image and the second is the name of the object. For example:

```
TV::image lookup 1|15 type "int *"
```

If no matching objects are found, the CLI returns an empty list. You can obtain a list of class objects by using the **lookup_keys** property.

properties — Lists the properties that the CLI can access. Do not use additional arguments with this option.

set	Sets the values of one or more image properties. The *other-args* argument contains property name and value pairs.
object-id	The ID of an image. An image ID is two integers that identify the base executable and an associated DLL. You can obtain a list of all image IDs by using the following command: `TV::image get –all id` If you use the **–all** option, TotalView carries out this operation on all images in the current focus.
other-args	Arguments required by the **get** and **set** subcommands.

Description:

The **TV::image** command lets you examine and set the image properties and states. These states and properties are:

id	The ID of the object.
is_dll	A true/false value where **1** indicates the image is a shared library and **0** if it is an executable.
lookup_keys	A list containing the object classes that can be used in a *by name* lookup; for example: `TV::image lookup 1|20 types foo` Currently, the only value returned is **{types}**.
name	The name of the image.
prototypes	A list of all of the prototypes that apply to an image.

Examples:

TV::image lookup 1|15 type "int *"

Finds the type identifiers for the **int *** type in image **1|15**. The result might be:

`1|25 1|76`

There can be more than one type with the same name in an image since many debugging formats provide separate type definitions in each source file.

```
foreach i [TV::image get –all id] {
        puts [format "%40s; %s" [TV::image get $i name] $i]}
```

Lists all current images along with their IDs.

```
set proto_id [TV::prototype create Array]
```

Creates a new array prototype.

```
TV::prototype set $proto_id                                    \
  name             {^(class|struct) (std::)?vector *<.*>$}    \
  language         C++                                        \
  validate_callback vector_validate                           \
  typedef_callback vector_typedef                             \
  type_callback    vector_type                                \
  rank_callback    vector_rank                                \
  bounds_callback  vector_bounds                              \
  address_callback vector_address
```

Sets properties for the prototype created in the previous example.

```
TV::image add 1|97 prototype $proto_id
```

Adds a prototype to image 1|97 so that it affects types defined in that image.

process

Sets and gets process properties

Format:

TV::process *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Lists the subcommands that you can use. The CLI responds by displaying the four subcommands shown here. Do not use other arguments with this subcommand.
- **get** — Gets the values of one or more process properties. The *other-args* argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the property names you entered.

  If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists the properties that the CLI can access. Do not use other arguments with this subcommand.
- **set** — Sets the values of one or more properties. The *other-args* arguments contains pairs of property names and values.

object-id — An identifier for a process. For example, **1** represents process 1. If you use the **–all** option, the subcommand is carried out on all objects of this class in the current focus.

other-args — Arguments required by the **get** and **set** subcommands.

Description:

The **TV::process** command lets you examine and set process properties and states. These states and properties are:

- **clusterid** — The ID of the cluster containing a process. This is a number uniquely identifying the TotalView server that owns the process. The ID for the cluster TotalView is running in is always **0** (zero).
- **duid** — The internal unique ID associated with an object.

executable	The program's name.
held	A value (either **1** or **0**) indicating if the process is held; **1** means that the process is held. (settable)
hostname	The name of the process's host system.
id	The process ID.
image_ids	A list of the IDs of all the images currently loaded into the process both statically and dynamically. The first element of the list is the current executable.
nodeid	The ID of the node upon which the process is running. The ID of each processor node is unique within a cluster.
state	Current state of the process. See **state_values** for a list of states.
state_values	Lists all possible values for the **state** property. These values can be **break**, **error**, **exited**, **running**, **stopped**, or **watch**.
syspid	The system process ID.
threadcount	The number of threads in the process.
threads	A list of threads in the process.

Examples:

TV::process get 3 threads

Gets the list of threads for process 3. For example:

```
1.1 1.2 1.4
```

TV::process get 1 image_ids

Returns a list of image IDs in process 1. For example:

```
1|1 1|2 1|3 1|4
```

f g TV::process get –all id threads

For each process in the group, creates a list with the process ID followed by the list of threads. For example:

```
{1 {1.1 1.2 1.4}} {2 {2.3 2.5}} {3 {3.1 3.7 3.9}}
```

```
foreach i [TV::process get 1 image_ids] {
   puts [TV::image get $i name] }
```

Prints the name of the executable and all shared libraries currently linked into the focus process. For example, the output of this command might be:

```
arraysAIX
/usr/lib/libxlf90.a
/usr/lib/libcrypt.a
/usr/lib/libc.a
```

prototype Sets and gets prototype properties

Format:

TV::**prototype** *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Lists the subcommands that you can use. The CLI responds by displaying the five subcommands shown here. Do not use other arguments with this subcommand.
- **create** — Creates a new prototype object. The *object-id* argument is not used; *other-args* is either **Array** or **Struct**, indicating the kind of prototype being created. You can change a prototype's properties up to the time you add it to an image. After being added, you can no longer change them.
- **get** — Gets the values of one or more prototype properties. The *other-args* argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the property names you entered.

  If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists the properties that the CLI can access. Do not use other arguments with this subcommand.
- **set** — Sets the values of one or more properties. The *other-args* argument consists of pairs of property names and values. The argument pairs that you can set are listed later in the section and are described in Chapter 5, "Type Transformations" on page 87.

object-id — The prototype ID. This value is returned when you create a new prototype. For example, **1** represents process 1. If you use the **–all** option, the subcommand is carried out on all objects of this class in the current focus.

other-args — Arguments required by the **get** and **set** subcommands.

Description:

The **TV::prototype** command lets you examine and set prototype properties and states. See Chapter 5, "Type Transformations" on page 87 for more information. These states and properties are:

address_callback Generates the address of the object's elements at run time. It returns either an absolute address, or an addressing expression that is appended to the address of the object to give the address of the field. (Returning an expression is the preferred method.)

For example, you might use a callback if the original data structure contains information on where the next data instance resides.

If you are creating a type for a distributed array, this procedure returns a two-element list. For more information, see "Distributed Arrays" on page 107.

The call structure for an address callback is:

address_callback *type_id object_addr index* [*replication*]

where:

type_id: is the type identifier for the type being prototyped.

object_addr: is the address of the object.

index: is the index string for the array element or the index of the field in the structure.

replication: is only used for distributed array objects. It indicates that the result is an address and an index into the distribution to determine the process within which this element resides, since this is a distributed array.

bounds_callback This is either a string that specifies the bounds statically in the format of the prototype's language, or a callback that TotalView calls when the object's address is known. If you are naming a callback, its call structure is:

bounds_callback *type_id object_addr*

This function returns a string in the format of the prototype's language that describes the current bounds; for example:

```
C: [2][40]
Fortran: (-2:10,-5:5)
```

If the bounds start with a bracket "[" or a parenthesis "(", TotalView assumes they are static; otherwise, TotalView assumes that the returned value is the name of a callback function.

As the bracket characters ([ ]) are special characters in Tcl, you must escape them even in strings; for example \[20\] rather than [20].

distribution_callback

Returns a list of process or thread identifiers that represent (in order) the processes/threads in which elements of this array exist. Only use this callback when your array is distributed over multiple processes. Its call structure is as follows:

distribution_callback *type_id object_addr*

TotalView calls this callback with a replication index. The returned value must have an index into the distribution as well as an address.

id — The prototype ID.

language — The language for this prototype. It indicates how TotalView parses and generates bounds and indices.

name — The regular expression that TotalView uses to match types. It must be anchored; that is, it must start with a "^" character.

rank_callback — Returns the rank of the array. TotalView calls this function when the prototype is modifying a type. Its call structure is as follows:

rank_callback *type_id*

type — The type of the prototype. This can be either **array** or **struct**.

Lists all possible values for **type**.

type_callback (required) TotalView invokes this callback when the prototype is modifying a type. Its format is:

type_callback *type_id*

The value returned depends on whether this is an array prototype or **struct** prototype.

A*rray prototypes*: Returns a value that is the type identifier for a single element of the array.

S*truct prototypes*: Returns a list in the format of the **struct_fields** property of a type that describes the struct type's fields. If a field in the type requires a callback, the addressing section of its field description should be the string **callback** rather than an addressing expression. In this case, TotalView uses **address_callback** to generate the address of this field.

typedef_callback Defines an *new_type_id* in terms of the *old_type_id*. You would use this callback when *old_type_id* is modified by this prototype. The format is:

typedef_callback *new_type_id old_type_id*

TotalView ignores any returned value.

validate_callback (required) TotalView invokes this callback whenever a type that matches the prototype's name is defined. It returns a Boolean value indicating if it should be applied. This callback lets you have more than one prototype match a type name, and then investigate the type to determine if TotalView should apply the prototype.

The call structure for a validation callback is:

validate_callback *type_id*

where *type_id* is the type identifier for the type being prototyped.

Description:

You will find an extensive discussion and examples in Chapter 5, "Type Transformations" on page 87.

quit

Terminates the debugging session

Format:

quit [–force]

Arguments:

–force	Tells the CLI that it should close all TotalView processes without asking permission.

Description:

The **quit** command terminates the CLI session.

The **exit** command terminates the TotalView session.

After executing the **quit** command, the CLI asks if it is all right to exit. If you answer yes, TotalView exits. If you had entered the CLI from the TotalView GUI, this command also closes the GUI window.

NOTE Type Ctrl-D to exit from the CLI window without exiting from TotalView.

Any processes and threads that were created by the CLI are destroyed. Any processes that existed prior to the debugging session (that is, were attached by the CLI as part of a **dattach** operation) are detached and left executing.

The **quit** and **exit** commands are interchangeable; they both do exactly the same thing.

Examples:

quit	Exits the CLI, leaving any "attached" processes running (in the run-time environment).

respond

Provides responses to commands

Format:

TV::respond *response command*

Arguments:

response	The response to one or more commands. If you include more than one response, separate the responses with newline characters.
command	One or more commands that the CLI will execute.

Description:

Executes a command. The *command* argument can be a single command or a list of commands. In most cases, you will place this information within braces ({}). If the CLI asks questions while *command* is executing, you are not asked for the answer. Instead, the CLI uses the characters in the *response* string for it. If more than question is asked and *response* is used up, TV::respond starts over at the beginning of the *response* string. If *response* does not end with a newline, TV::respond appends one.

Do not use this command to suppress the MORE prompt in macros. You should instead use the following command:

```
dset LINES_PER_SCREEN 0
```

The most common values for response are y and n.

NOTE If you are using the TotalView GUI and the CLI at the same time, your CLI command may cause a dialog boxes to appear. You cannot use the TV::respond command to close or interact with these dialog boxes.

Examples:

```
TV::response {y} {exit}
```

Exits from TotalView. This command automatically answers the "Do you really wish to exit TotalView" question.

```
set f1 y
set f2 exit
TV::response $f1 $f2
```

A way to exit from TotalView without seeing the "Do you really wish to exit TotalView" question. Neither of these two uses is recommended. Instead, you can use "exit –force".

stty

Sets terminal properties

Format:

stty [*stty-args*]

Arguments:

stty-args	One or more UNIX **stty** command arguments as defined in the **man** page for your operating system.

Description:

The CLI **stty** command executes a UNIX **stty** command on the **tty** associated with the CLI window. This lets you set all of your terminal's properties. However, this is most often used to set erase and kill characters.

If you start the CLI from a terminal by using the **totalviewcli** command, the **stty** command alters this terminal's environment. Consequently, the changes you make using this command are retained within the terminal after you exit.

If you omit *stty-args*, the CLI displays information describing your current settings.

The output from this command is returned as a string.

Examples:

stty	Prints information about your terminal settings. The information printed is the same as if you had entered **stty** while interacting with a shell.
stty –a	Prints information about all of your terminal settings.
stty erase ^H	Sets the *erase* key to Backspace.
stty sane	Resets the terminal's settings to values that the shell thinks they should be. If you are having problems with command-line editing, use this command. (The **sane** option is not available in all environments.)

source_process_startup

"Sources" a .tvd file when a process is loaded

Format:

TV::source_proccess_startup *process_id*

Arguments:

process_id	The PID of the current process.

Description:

The TV::**source_process_startup** command loads and interprets the **.tvd** file associated with the current process. That is, if a file named *executable*.**tvd** exists, the CLI *sources* it.

thread

Gets and sets thread properties

Format:

TV::thread *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Lists the subcommands that you can use. The CLI responds by displaying the four subcommands shown here. Do not use other arguments with this option.
- **get** — Gets the values of one or more thread properties. The *other-args* argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the property names you entered.

  If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists an object's properties. Do not use other arguments with this option.
- **set** — Sets the values of one or more properties. The *other-args* argument contains paired property names and values.

object-id — A thread ID. If you use the **–all** option, the operation is carried out on all threads in the current focus.

other-args — Arguments required by the **get** and **set** subcommands.

Description:

The **TV::thread** command lets you examine and set the thread properties and states. These states and properties are:

- **continuation_sig** — The signal that should be passed to a thread the next time it runs. On some systems, the thread receiving the signal may not always be the one for which this property was set.
- **dpid** — The ID of the process associated with a thread.
- **duid** — The internal unique ID associated with the thread.

held	A value (either **1** or **0**) indicating if the thread is held; **1** means that the thread is held. (settable)
id	The ID of the thread.
manager	A value (either **1** or **0**) indicating if this is a system manger thread; **1** means that it is.
pc	Current PC at which the target is executing. (settable)
state	Current state of the target. See **state_values** for a list of states.
state_values	A list of values for the **state** property. These values are **break**, **error**, **exited**, **running**, **stopped**, and **watch**.
systid	The system thread ID.

Examples:

f p3 TV::thread get –all id

Returns a list of thread IDs for process 3. For example:

```
1.1 1.2 1.4
```

type

Gets and sets type properties

Format:

TV::type *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Lists the subcommands that you can use. The CLI responds by displaying the four subcommands shown here. Do not use other arguments with this option.
- **get** — Gets the values of one or more type properties. The *other-args* argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the property names you entered.

  If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists a type's properties. Do not use other arguments with this option.
- **set** — Sets the values of one or more type properties. The *other-args* argument contains paired property names and values.

object-id — An identifier for an object. For example, **1** represents process 1, and **1.1** represents thread 1 within process 1. If you use the **–all** option, the operation is carried out on all objects of this class in the current focus.

other-args — Arguments required by the **get** and **set** subcommands.

Description:

The **TV::type** command lets you examine and set the type properties and states. These states and properties are:

- **enum_values** — For an enumerated type, a list of **{name value}** pairs giving the definition of the enumeration. If you apply this to a non-enumerated type, the CLI returns an empty list.
- **id** — The ID of the object.

image_id	The ID of the image in which this type is defined.
language	The language of the type.
length	The length of the type.
name	The name of the type; for example, class foo.
prototype	The ID for the prototype. If the object is not prototyped, the returned value is **{}**.
rank	(array types only) The rank of the array.
struct_fields	(class/struct/union types only). A list of lists giving the description of all the type's fields. Each sublist contains the following fields: { *name type_id addressing properties* } where: *name* is the name of the field. *type_id* is simply the *type_id* of the field. *addressing* contains additional addressing information that points to the base of the field. *properties* contains an additional list of properties in the following format: **"[virtual] [public|private|protected] base class"** If no properties apply, this string is null. If you use **get struct_fields** for a type that is not a class, struct, or a union, the CLI returns an empty list.
target	For an array or pointer type, returns the ID of the array member or target of the pointer. If this is not applied to one of these types, the CLI returns an empty list.
type	Returns a string describing this type. For example, **signed integer**.
type_values	Returns all possible values for the **type** property.

Examples:

TV::type get 1|25 length target

Finds the length of a type and (assuming it is a pointer or an array type) the target type. The result may look something like:

```
                    4 1|12
```

The following example uses the **TV::type properties** command to obtain the list of properties:

```
d1.<> \
proc print_type {id} {
    foreach p [TV::type   properties] {
        puts [format "%13s  %s" $p [TV::type get $id $p]]
    }
} d1.<> print_type 1|6

    enum_values
             id  1|6
       image_id  1|1
       language  f77
  len.DefShort   4
           Body{  <integer>

       margin-   0
    top:0.0in;

       margin-   Signed Integer
    left:.65in;  {Array} {Array of characters} {Enumeration}...

              }

            gth
           name
      prototype
           rank
  struct_fields
         target
           type
    type_values

d1.<>
```

unalias

Removes a previously defined alias

Format:

Removes an alias

unalias *alias-name*

Removes all aliases

unalias –all

Arguments:

alias-name	The name of the alias being deleted.
–all	Tells the CLI to remove all aliases.

Description:

The **unalias** command removes a previously defined alias. You can delete all aliases by using the **–all** option. Aliases defined in the **tvdinit.tvd** file are also deleted.

Examples:

unalias step2	Removes the **step2** alias; **step2** is now undefined and can no longer be used. If **step2** was included as part of the definition of another command, that command will no longer work correctly. However, the CLI will only display an error message when you try to execute the alias that contains this removed alias.
unalias –all	Removes all aliases.

Appendix A

CLI Command Summary

This appendix contains a summary of all CLI commands.

actionpoint

Gets and sets action point properties

TV::actionpoint *action* [*object-id*] [*other-args*]

alias

Creates or views pseudonyms for commands

alias *alias-name defn-body*

Views previously defined aliases

alias [*alias-name*]

capture

Returns a command's output as a string

capture *command*

dactions

Displays information, saves, and reloads action points

dactions [*ap-id-list*] [**–at** *source-loc*]
[**–enabled** | **–disabled**]

Saves action points to a file

dactions -save [*filename*]

Loads previously saved action points

dactions -load [*filename*]

dassign

Changes the value of a scalar variable

dassign *target value*

dattach

Brings currently executing processes under CLI control

dattach [-g *gid*] [-r *hname*]
[-ask_attach_parallel | -no_attach_parallel]
[-e] *fname pid-list*

dbarrier

Creates a barrier breakpoint at a source location

dbarrier *source-loc* [-stop_when hit { group | process | none }]
[-stop_when_done { group | process | none }]

Creates a barrier breakpoint at an address

dbarrier -address *addr*
[-stop_when_hit { group | process | none }]
[-stop_when_done { group | process | none }]

dbreak

Creates a breakpoint at a source location

dbreak *source-loc* [–p | –g | -t] [[–l *lang*] –e *expr*]

Creates a breakpoint at an address

dbreak –address *addr* [–p | –g | -t] [[–l *lang*] –e *expr*]

dcheckpoint

Creates a checkpoint image of processes (SGI only)

dcheckpoint [*after_checkpointing*] [-by *process_set*] [-no_park]
[-ask_attach_parallel | -no_attach_parallel]
[-no_preserve_ids] [-force] *checkpoint-name*

dcont

Continues execution and waits for execution to stop

dcont

ddelete

Deletes some action points

ddelete *action-point-list*

Deletes all action points

ddelete –a

ddetach

Detaches from the processes

ddetach

ddisable

Disables some action points

ddisable *action-point-list*

Disables all action points

ddisable –a

ddown

Moves down the call stack

ddown [*num-levels*]

dec2hex

Converts a decimal number into hexadecimal

TV::dec2hex *number*

denable

Enables some action points

denable *action-point-list*

Enables all disabled action points in the current focus

denable –a

dfocus

Changes the target of future CLI commands to this P/T set

dfocus *p/t-set*

Executes a command within this P/T set

dfocus *p/t-set command*

dgo

Resumes execution of target processes

dgo

dgroups

Adds members to thread and process groups

dgroups -add [**-g** *gid*] [*id-list*]

Deletes groups

dgroups -delete [**-g** *gid*]

Intersects a group with a list of processes and threads

dgroups -intersect [**-g** *gid*] [*id-list*]

Prints process and thread group information

dgroups [**-list**] [*pattern*]

Creates a new thread or process group

dgroups -new [*thread_or_process*] [**-g** *gid*] [*id-list*]

Removes members for thread or process groups

dgroups -remove [**-g** *gid*] [*id-list*]

dhalt

Suspends execution of processes

dhalt

dhold

Holds processes

dhold -process

Holds threads

dhold -thread

dkill

Terminates execution of target processes

dkill

dlappend

Appends list elements to a TotalView variable

dlappend *variable-name value* [...]

dlist

Displays code relative to the current list location

dlist [**–n** *num-lines*]

Displays code relative to a named location

dlist *source-loc* [**–n** *num-lines*]

Displays code relative to the current execution location

dlist –e [**–n** *num-lines*]

dload

Loads debugging information

dload [**-g** *gid*] [**-r** *hname*] [**-e**] *executable*

dnext

Steps source lines, stepping over subroutines

dnext [*num-steps*]

dnexti

Steps machine instructions, stepping over subroutines

dnexti [*num-steps*]

dout

Runs out from current subroutine

dout [*frame-count*]

Runs until the PC returns into a procedure

dout *proc-name*

dprint

Prints the value of a variable

dprint *variable*

Prints the value of an expression

dprint *expression*

dptsets

Shows status of processes and threads in an array of P/T expressions

dptsets [*ptset_array*] ...

drerun

Restarts processes

drerun [*args*]

drestart

Restarts a checkpoint

drestart [*process-state*] [**-no_unpark**] [**-g** *gid*] [**-r** *host*]
[**-ask_attach_parallel** | **-no_attach_parallel**]
[**-no_preserve_ids**] *checkpoint-name*

drun

Starts or restarts processes

drun [*cmd_arguments*] [< *infile*]
[> [>][&] *outfile*]
[2> [>] *errfile*]

dset

Creates or changes a CLI state variable

dset [**-new**] *debugger-var value*

Views current CLI state variables

dset [*debugger-var*]

source_process_startup

"Sources" a .tvd file when a process is loaded

TV::source_proccess_startup *process_id*

dstatus

Shows current status of processes and threads

dstatus

dstep

Steps lines, stepping into subfunctions

dstep [*num-steps*]

dstepi

Steps machine instructions, stepping into subfunctions

dstepi [*num-steps*]

dunhold

Releases a process

dunhold -process

Releases a thread

dunhold -thread

dunset

Restores a CLI variable to its default value

dunset *debugger-var*

Restores all CLI variables to their default values

dunset –all

duntil

Runs to a line

duntil *line-number*

Runs to an address

duntil -address *addr*

Runs into a function

duntil *proc-name*

dup

Moves up the call stack

dup [*num-levels*]

dwait

Blocks command input until the target processes stop

dwait

dwatch

Defines a watchpoint for a variable

dwatch *variable* [–length *byte-count*] [–p | –g | –t]
[[–l *lang*] –e *expr*] [–t *type*]

Defines a watchpoint for an address

dwatch –address *addr* –length *byte-count* [–p | –g | –t]
[[–l *lang*] –e *expr*] [–t *type*]

dwhat

Determines what a name refers to

dwhat *symbol-name*

dwhere

Displays locations in the call stack

dwhere [*num-levels*] [–args]

Displays all locations in the call stack

dwhere –a [–args]

dworker

Adds or removes a thread from a workers group

dworker { *number* | *boolean* }

errorCodes

Returns a list of all error code tags

TV::errorCodes

Returns or raises error information

TV::errorCodes *number_or_tag* [-raise [*message*]]

exit

Terminates the debugging session

exit [–force]

focus_groups

Returns a list of groups in the current focus

TV::focus_groups

focus_processes
Returns a list of processes in the current focus

TV::focus_processes [-all | -group | -process | -thread]

focus_threads
Returns a list of threads in the current focus

TV::focus_threads [-all | -group | -process | -thread]

group
Gets and sets group properties

TV::group *action* [*object-id*] [*other-args*]

help
Displays help information

help [*topic*]

hex2dec
Converts to decimal

TV::hex2dec *number*

image
Gets and sets image properties

TV::image *action* [*object-id*] [*other-args*]

process
Gets and sets process properties

TV::process *action* [*object-id*] [*other-args*]

prototype
Gets and sets prototype properties

TV::prototype *action* [*object-id*] [*other-args*]

quit
Terminates the debugging session

quit [–force]

respond
Provides responses to commands

TV::respond *response command*

stty

Sets terminal properties

stty [*stty-args*]

thread

Gets and sets thread properties

TV::thread *action* [*object-id*] [*other-args*]

type

Gets and sets type properties

TV::type *action* [*object-id*] [*other-args*]

unalias

Removes an alias

unalias *alias-name*

Removes all aliases

unalias –all

Appendix B

CLI Command Default Arena Widths

This appendix lists all CLI commands and their default arena widths.

Command	Default Arena Width
actionpoint	—
alias	—
capture	—
dactions	process
dassign	thread; if the current width is "process", **dassign** acts on each thread in the process
dattach	—
dbarrier	Obtains focus from the setting of the SHARE_ACTION_POINT variable; if true, the default is "group"; if false, the default is "process"
dbreak	Obtains focus from the setting of the SHARE_ACTION_POINT variable; if true, the default is "group"; if false, the default is "process"
dcheckpoint	process
dcont	process
ddelete	process
ddetach	process
ddisable	process
ddown	thread; if the current width is "process", **ddown** acts on each thread in the process
dec2hex	—

Command	Default Arena Width
denable	process
dfocus	—
dgo	process
dgroups	The **-list** option ignores the focus; other options use group width to find the groups being operated upon and thread width to find the operands
dhalt	process
dhold	depends on the **-thread** or -**process** option
dkill	process; note that killing the primary process for a group always kills all of its slaves
dlappend	—
dlist	thread; if the current width is "process", **dlist** iterates over all threads in the process
dload	—
dnext	process
dnexti	process
dout	process
dprint	thread; if the current width is "process", **dprint** acts on each thread in the process
dptsets	—
drerun	process
drestart	—
drun	process
dset	—
dstatus	process
dstep	process
dstepi	process
dunhold	depends on the **-thread** or -**process** option
dunset	—
duntil	process

Command	Default Arena Width
dup	thread; if the current width is "process", **dup** acts on each thread in the process
dwait	process
dwatch	Obtains focus from the **SHARE_ACTION_POINT** variable's setting **true**: default to "group" **false**: default to "process"
dwhat	thread; if the current width is "process", **dwhat** acts on each thread in the process
dwhere	thread; if the current width is "process", **dwhere** acts on each thread in the process
dworker	thread
errorCodes	—
exit	—
focus_groups	group
focus_processes	process
focus_threads	thread
group	—
help	—
hex2dec	—
process	—
prototype	—
quit	—
respond	—
stty	—
thread	—
type	—
unalias	—

Appendix C

Distributed Array Type Mapping

This appendix contains listings of two files. The first, mandel.c, is an OpenMP program that creates a Mandelbrot set. The second, cyclic_array.tcl, implements a type mapping that allows TotalView to display the distributed array created by mandel.c in one Variable Window.

The mandel.c Program

The following program is a simple Mandelbrot program that uses a distributed array. This program is not a good Mandelbrot program as its purpose is to demonstrate a method for handling a cyclically distributed array and how TotalView can reconstruct the global array from the distributed components. Filling the array with the Mandelbrot set simply makes it easier to see that everything is working.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

/*
 * This represents a square two dimensional array, where the
 * first index is cyclically distributed over the processors.
 */
struct cyclic_array
{
    int * local_elements;
    int local_count;
    int global_count;
```

```
    int numprocs;
    int myproc;
};

/*
 * Set up the array.
 */
void setup_array (struct cyclic_array *a, int count, int np, int
me)
{
    a->numprocs = np;
    a->myproc   = me;

    a->local_count = count/np + 1;
    a->global_count= count;

    a->local_elements = malloc (sizeof(int)
                * count * a->local_count);
}

/*
 * Compute the owning processor.
 */
int owner_of (struct cyclic_array * a, int ix)
{
    return ix%(a->numprocs);
}

/*
 * Compute the address of the local column given the global
 * column's index.
 */
int * local_column (struct cyclic_array * a, int ix)
{
    int owner = owner_of (a, ix);

    if (owner != a->myproc)
        return (int *)0;

    return a->local_elements + (ix/a->numprocs)
                * a->global_count;
}
```

```
/*
 * Functions for handling complex numbers
 */
struct complex
{
    float re;
    float im;
};

float cabs_squared (struct complex * z)
{
    return z->re*z->re + z->im*z->im;
}

struct complex
ctimes (struct complex * z0, struct complex * z1)
{
    struct complex res;

    res.re = z0->re * z1->re - z0->im * z1->im;
    res.im = z0->re * z1->im + z0->im * z1->re;

    return res;
}

struct complex
cplus (struct complex * z0, struct complex * z1)
{
    struct complex res;

    res.re = z0->re + z1->re;
    res.im = z0->im + z1->im;

    return res;
}

/*
 * Compute the number of iterations needed to determine that
 * the value is outside the Mandelbrot set.
 */
int mandel (struct complex z0)
{
```

```
    struct complex z = z0;
    int i;

    for (i=0; i<31; i++)
    {
        if (cabs_squared (&z) >= 4.0)
            break;

        /* The Mandelbrot step, z = z*z + z0 */
        z = ctimes (&z, &z);
        z = cplus (&z, &z0);
    }

    return i;
}

/* Convert an index into a scaled position in -2..2 */
float position (struct cyclic_array * a, int idx)
{
    idx = idx-(a->global_count/2);

    return 4*((0.5 + (float) idx)/a->global_count);
}

/*
 * Collect the distributed array and print it.
 * May be rotated...
 */
void print_array (struct cyclic_array * a)
{
    int ix;
    int * buffer = malloc (sizeof (int) * a->global_count);

    for (ix = 0; ix<a->global_count; ix++)
    {
        int * col = local_column (a, ix);
        MPI_Status stat;

        if (a->myproc == 0)
        { /* First process does the printing */
            int iy;
```

```
            if (col == 0)
            {
                MPI_Recv (buffer, a->global_count, MPI_INT,
                        MPI_ANY_SOURCE, ix,
                        MPI_COMM_WORLD, &stat);
                col = buffer;
            }

            for (iy=0; iy<a->global_count; iy++)
            {
                printf ("%2d ", col[iy]);
            }
            printf ("\n");
        }
        else
        { /* All others send the data */
            if (col != 0)
            {
                MPI_Send (col, a->global_count, MPI_INT, 0,
                        ix, MPI_COMM_WORLD);
            }
        }
    }

    MPI_Barrier (MPI_COMM_WORLD);
    free (buffer);
}

/* We could read this from the user if we wanted to */
#define GLOBAL_SIZE 40

/*
 * Finally we can do the business !
 */
int main (int argc, char ** argv)
{
    int myrank;
    int nprocs;
    struct cyclic_array a;
    int ix;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    setup_array (&a, GLOBAL_SIZE, nprocs, myrank);

    /* For each column we own calculate the results */
    for (ix = 0; ix < GLOBAL_SIZE; ix++)
    {
        int * lp = local_column (&a, ix);

        if (lp != 0)
        {
            struct complex c;
            int iy;

            c.re = position (&a, ix);

            for (iy = 0; iy < GLOBAL_SIZE; iy++)
            {
                c.im = - position (&a, iy);/* - to flip it vertically */

                lp [iy] = mandel (c);
            }
        }
    }

    print_array (&a);
}
```

The cyclic_array.tcl Type Mapping File

The following listing shows the callback and utility procedures as well as the **TV::prototype** commands needed to implement a type mapping that allows you to view a distributed array.

```
# Check that the type is what we expect, and that we can
# locate the appropriate fields.
#
# We're expecting
#   struct cyclic_array
#   {
```

```
#       int * local_elements;
#       int local_count;
#       int global_count;
#       int numprocs;
#       int myproc;
#   };

proc da_validate {instance_id} {
    global _da_info

    set fields [TV::type get $instance_id struct_fields]
    #
    # We'll save four properties of each type:
    #   The offset of the pointer to the local array.
    #   The target type identifier.
    #   The target type size
    #   The offset of the local_count
    #   The offset of the global count
    set typeinfo [list {} {} {} {} {}]
    set matched 0

    foreach field $fields {
        set name [lindex $field 0]
        set addressing [lindex $field 2]

        switch -- $name {
            local_elements  {
                set typeinfo [lreplace $typeinfo 0 0 \
                        [extract_offset $addressing]]
                # Extract the target type too.
                set field_typeid \
                        [TV::type get [lindex $field 1] target]

                set typeinfo [lreplace $typeinfo 1 1 $field_typeid]
                set typeinfo [lreplace $typeinfo 2 2 \
                        [TV::type get $field_typeid length]]

                incr matched
            }

            local_count {
                set typeinfo [lreplace $typeinfo 3 3 \
                        [extract_offset $addressing]]
```

```
                incr matched
            }

            global_count {
                set typeinfo [lreplace $typeinfo 4 4 \
                        [extract_offset $addressing]]
                incr matched
            }
        }
    }

    if {$matched != 3} {
        return false
    }

    set _da_info($instance_id) $typeinfo
    return true;
}


#
# Copy any properties we require when defining a new type as a
# typedef for a type which already has this prototype.
#
proc da_typedef {new_id old_id} {
    global _da_info
    set _da_info($new_id) $_da_info($old_id)
}

#
# Return the target type for this array.
#
proc da_type {instance_id} {
    global _da_info
    set typeinfo $_da_info($instance_id)

    return [lindex $typeinfo 1]
}

#
# It's a two-dimensional array.
#
proc da_rank {type_id} {
```

```
    return 2
}

#
# Compute the bounds of the required element of the array.
#
proc da_bounds {type_id address} {
    global _da_info
    set typeinfo $_da_info($type_id)

    set address [expr $address + [lindex $typeinfo 4]]
    set bound [read_store $address int]

    return "\[$bound\]\[$bound\]"
}

#
# Compute the address of the required element of the array.
#
proc da_address {type_id address indices replication} {
    #
    # Each element lives in only one place, so we return a null
    # result if asked for other places for it.
    if {$replication != 0} {
        return ""
    }

    global _da_info _da_nprocs

    set typeinfo $_da_info($type_id)
    set bound [read_store \
                    [expr $address + [lindex $typeinfo 4]] int]

    set distributed_index [lindex $indices 0]
    set other_index [lindex $indices 1]
    set node [expr $distributed_index%$_da_nprocs]
    set local_index [expr $distributed_index/$_da_nprocs]

    set element_size [lindex $typeinfo 2]

    #
    # We have to work out the whole address.
    #
```

```
    set delta [expr $element_size* \
                ($other_index+$bound*$local_index)]

    return \
        "$node {addc [lindex $typeinfo 0]; indirect; addc $delta}"
}

#
# Return the list of process/thread identifiers over which this
# array is distributed. In this simple example, any of these
# arrays are distributed over all the processes.

proc da_distribution {type_id address} {
    #
    # For the moment we assume this is all items in our
    # workers group.
    global GROUP WGROUP _da_nprocs

    # Choose the first process in the focus set.
    set proc [lindex [TV::focus_processes] 0]

    # Find the relevant worker group identifier.
    set group_id $WGROUP($proc)

    # Extract the member identifiers from the worker
    # contents.
    set res [lrange $GROUP($group_id) 1 end]

    # Save the number of processes for later.
    set _da_nprocs [llength $res]

    return $res
}
```

Glossary

ACTION POINT: A debugger feature that allows a user to request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, evaluation points, and barriers.

ACTION POINT IDENTIFIER: A unique integer ID associated with an action point.

ADDRESS SPACE: A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

AFFECTED P/T SET: The set of process and threads that will be affected by the command. For most commands, this is identical to the target P/T set, but in some cases it may include additional threads. (See "P/T (process/thread) set" on page 299 for more information.)

AGGREGATED OUTPUT: The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

ARENA: A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

ASYNCHRONOUS: IWhen processeses communicate with one another, they send messages. If a process decides that it does not want to wait for an answer, it is said to run "asynchronously". For example, in most client/server programs, one program sends an RPC request to a second program and then waits to receive a response from the second program. This is the nor-

mal *synchronous* mode of operation. If, however, the first program sends a message and then continues executing, not waiting for a reply, the first mode of operations is said to be *asynchronous*.

AUTOMATIC PROCESS ACQUISITION: TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you do not have to attach to them manually. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition automatically starts the TotalView Debugger Server (the **tvdsvr**).

BARRIER: An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

BASE WINDOW: The original Process Window or Variable Window before you dive into routines or variables. After diving, you can use a **Reset** or **Undive** command to restore this original window.

BLOCKED: A thread state where the thread is no longer executing because it is wating for an event to occur. In most cases, the thread is blocked because it is waiting for a mutex or condition state.

BREAKPOINT: A point in a program where execution can be suspended to permit examination and manipulation of data.

CALL STACK: A higher-level view of stack memory, interpreted in terms of source program variables and locations.

CHILD PROCESS: A process created by another process (*see* "parent process" on page 298) when that other process calls **fork()**.

CLOSED LOOP: see **closed loop**.

CLUSTER DEBUGGING: The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.

COMMAND HISTORY LIST: A debugger-maintained list storing copies of the most recent commands issued by the user.

CONDITION SYNCHRONIZATON: A process that delays thread execution until a condition is satisfied.

CONTEXTUALLY QUALIFIED (SYMBOL): A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process identifier, thread identifier, frame number, and variable or subprocedure name.

CORE FILE: A file containing the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

CORE-FILE DEBUGGING: A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

CROSS-DEBUGGING: A special case of remote debugging where the host platform and the target platform are different types of machines.

CURRENT FRAME: The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

CURRENT LANGUAGE: The source code language used by the file containing the current source location.

CURRENT LIST LOCATION: The location governing what source code will be displayed in response to a list command.

DATASET: A set of array elements generated by TotalView and sent to the Visualizer. (See **"visualizer process" on page 304**.)

DBELOG LIBRARY: A library of routines for creating event points and generating event logs from within TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

DBFORK LIBRARY: A library of special versions of the **fork()** and **execve()** calls used by the TotalView debugger to debug multiprocess programs. If you link your program with TotalView's **dbfork** library, TotalView will be able to automatically attach to newly spawned processes.

DEBUGGING INFORMATION: Information relating an executable to the source code from which it was generated.

DEBUGGER INITIALIZATION FILE: An optional file establishing initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called **.tvdrc**.

DEBUGGER PROMPT: A string printed by the CLI that indicates that it is ready to receive another user command.

DEBUGGER SERVER: *See* **tvdsvr process**.

DEBUGGER STATE: Information that TotalView or the CLI maintains in order to interpret and respond to user commands. Includes debugger modes, user-defined commands, and debugger variables.

DISTRIBUTED DEBUGGING: The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message-passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PARMACS) run on more than one host.

DIVE STACK: A series of nested dives that were performed in the same variable window. The number of greater-than symbols (>) in the upper left-hand corner of a Variable Window indicates the number of nested dives on the dive stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of greater-than symbols shown in the Variable Window.

DIVING: The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

DOPE VECTOR: This is a runtime descriptor that contains all information about an object that requires more information than is available as a single pointer or value. For example, you might declare a Fortran 90 pointer variable that is a pointer to some other object but which has its own upper bound as follows:

integer, pointer, dimension (:) :: iptr

Assume that you initialize it as follows:

iptr => iarray (20:1:-2)

iptr is now a synonym for every other element in the first twenty elements of **iarray** and this pointer array is in reverse order. For example, **iptr(1)** maps to **iarray(20)**, **iptr(2)** maps to **iarray(18)**, and so on.

A compiler represents an **iptr** object using a run time descriptor) that contains (at least) elements such as a pointer to the first element of the actual data, a stride value, and a count of the number of elements (or equivalently an upper bound).

DPID: Debugger ID. This is the ID TotalView uses for processes.

EDITING CURSOR: A black rectangle that appears when a TotalView GUI field is selected for editing. You use field editor commands to move the editing cursor.

EVALUATION POINT: A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

EVENT LOG: A file containing a record of events for each process in a program.

EVENT POINT: A point in the program where TotalView writes an event to the event log for later analysis with TimeScan.

EXECUTABLE: A compiled and linked version of source files, containing a "main" entry point.

EXPRESSION: An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of the current source language. Not all Fortran 90, C, and C++ operators are supported.

EXTENT: The number of elements in the dimension of an array. For example, a Fortran array of integer(7,8) has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

FIELD EDITOR: A basic text editor that is part of TotalView's interface. The field editor supports a subset of GNU Emacs commands.

FOCUS: The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you are using the default prompt).

FRAME: An area in stack memory containing the information corresponding to a single invocation of a subprocedure.

FULLY QUALIFIED (SYMBOL): A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

GID: The TotalView group ID.

GOI: The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and the like.

GRIDGET: A dotted grid in the tag field that indicates you can set an action point on the instruction.

GROUP: When TotalView starts processes, it places related processes in families. These families are called "groups."

GROUP OF INTEREST: The primary group that is affected by a command.

HEAP: An area of memory that your program uses when it dynamically allocates blocks of memory. It is also how people describe my car.

HOST MACHINE: The machine on which the TotalView debugger is running.

INITIAL PROCESS: The process created as part of a load operation, or that already existed in the run-time environment and was attached by TotalView or the CLI.

INFINITE LOOP: See **loop, infinite**.

LVALUE: A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

LHS EXPRESSION: This is a synonym for **lvalue**.

LOOP, INFINITE: see **infinite loop**.

LOWER BOUND: The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

MACHINE STATE: Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

MESSAGE QUEUE: A list of messages sent and received by message-passing programs.

MPICH: MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see www.mcs.anl.gov/mpi.

MPMD (MULTIPLE PROGRAM MULTIPLE DATA) PROGRAMS: A program involving multiple executables, executed by multiple threads and processes.

MUTEX (MUTUAL EXCLUSION): Techniques for sharing resources so that different users do not conflict and cause unwanted interactions.

NATIVE DEBUGGING: The action of debugging a program that is running on the same machine as TotalView.

NESTED DIVE: TotalView lets you dive into pointers, structures, or arrays within a variable. When you dive into one of these elements, TotalView updates the display so that the new element is displayed. So, a nested dive is a *dive* within a dive. You can return to the previous display by selecting the left-facing arrow in the top right corner of the window.

NODE: A machine on a network. Each machine has a unique network name and address.

OUT OF SCOPE: When symbol lookup is performed for a particular symbol name and it is not found in the current scope or any containing scopes, the symbol is said to be out of scope.

PARALLEL PROGRAM: A program whose execution involves multiple threads and processes.

PARALLEL TASKS: Tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results. (*llnl*)

PARALLELIZABLE PROBLEM: A problem that can be divided into parallel tasks. This may require changes in the code and/or the underlying algorithm. (*llnl*)

PARCEL: The number of bytes required to hold the shortest instruction for the target architecture.

PARENT PROCESS: A process that calls **fork()** to spawn other processes (usually called "child processes").

PARMACS LIBRARY: A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

PARTIALLY QUALIFIED (SYMBOL): A symbol name that includes only some of the levels of source code organization (for example, filename and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

PC: This is an abbreviation for P*rogram Counter.*

PID: Depending on context, this is either the "process ID" or the the "program ID". In most cases, this will be a process ID.

POI: The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and the like.

PROCESS: An executable that is loaded into memory and is running (or capable of running).

PROCESS GROUP: A group of processes associated with a multiprocess program. A process group includes program control groups and share groups.

PROCESS/THREAD IDENTIFIER: A unique integer ID associated with a particular process and thread.

PROCESS OF INTEREST: The primary process that is affected by a command.

PROGRAM EVENT: A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

PROGRAM CONTROL GROUP: A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent) and children that were forked with a subsequent call to **execve()** (processes that do *not* share the same source code as the parent). Contrast with **share group**.

PROGRAM STATE: A higher-level view of the machine state, where addresses, instructions, registers, and such, are interpreted in terms of source program variables and statements.

P/T (PROCESS/THREAD) SET: The set of threads drawn from all threads in all processes of the target program.

PVM LIBRARY: Parallel Virtual Machine library. A message-passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.

RACE CONDITION: A problem that occurs when threads try to simultaneously access a resource. The result can be a deadlock, data corruption, or a program fault.

REMOTE DEBUGGING: The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

RESUME COMMANDS: Commands that cause execution to restart from a stopped state: **dstep**, **dgo**, **dcont**, **dwait**.

RHS EXPRESSION: This is a synonym for **rvalue**.

RVALUE: An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

SATISFACTION SET: The set of processes and threads that must be held before a barrier can be satisfied.

SATISFIED: A condition indicating that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are

either running because they have not yet hit the barrier or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is *satisfied*, the held processes or threads are released, which means they can now be run. Prior to this event, they could not be run.

SERIAL EXECUTION: Execution of a program sequentially, one statement at a time. (*llnl*)

SERIAL LINE DEBUGGING: A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.

SHARE GROUP: A group of processes that includes the parent process and any related processes that share the same source code as the parent. Contrast with **program control group**.

SHARED LIBRARY: A compiled and linked set of source files that are dynamically loaded by other executables—and have no "main" entry point.

SIGNALS: Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

SINGLE STEP: The action of executing a single statement and stopping (as if at a breakpoint).

SLICE: A subsection of an array, which is expressed in terms of a lower bound, upper bound, and stride. Displaying a slice of an array can be useful when working with very large arrays, which is often the case in Fortran programs.

SOURCE FILE: Program file containing source language statements. TotalView allows you to debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler.

SOURCE LOCATION: For each thread, the source code line it will execute next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

SPAWNED PROCESS: The process created by a user process executing under debugger control.

SPMD (SINGLE PROGRAM MULTIPLE DATA) PROGRAMS: A program involving just one executable, executed by multiple threads and processes.

STACK: A portion of computer memory and registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

STACK FRAME: A section of the stack that contains the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the program counter (PC) at the time the routine was called.

STACK POINTER: A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored.

STACK TRACE: A sequential list of each currently active routine called by a program and the frame pointer pointing to its stack frame.

STATIC (SYMBOL) SCOPE: A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

STEPPING: Advancing program execution by fixed increments, such as by source code statements.

STOP SET: A set of threads that should be stopped once an action point has been triggered.

STOPPED/HELD STATE: The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) will be required before it is capable of continuing execution.

STOPPED/RUNNABLE STATE: The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

STOPPED STATE: The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

STRIDE: The interval between array elements in a slice and the order in which the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is –1, every element between the upper bound and lower bound (reverse order) is displayed.

SYMBOL: Entities within program state, machine state, or debugger state.

SYMBOL LOOKUP: Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively so that containing scopes are searched in an outward progression.

SYMBOL NAME: The name associated with a symbol known to TotalView (for example, function, variable, data type, and such).

SYMBOL TABLE: A table of symbolic names (such as variables or functions) used in a program and their memory locations. The symbol table is part of the executable object generated by the compiler (with the **–g** option) and is used by debuggers to analyze the program.

SYNCHRONIZATION: A mechanism that prevents problems caused by concurrent threads manipulating shared resources. The two most common mechanisms for synchronizing threads are mutual exclusion and condition synchronization.

TAG FIELD: The left margin in the Source Pane of the TotalView Process Window containing boxed line numbers marking the lines of source code that actually generate executable code.

TARGET MACHINE: The machine on which the process to be debugged is running.

TARGET PROCESS SET: The target set for those occasions when operations can only be applied to entire processes, not to individual threads within a process.

TARGET PROGRAM: The executing program that is the target of debugger operations.

TARGET P/T SET: The set of processes and threads upon which a CLI command will act.

TASK: A logically discrete section of computational work. (This is an informal definition.) (*llnl*)

THREAD: An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

THREAD EXECUTION STATE: The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of predefined states.

THREAD OF INTEREST: The primary thread that will be affected by a command.

TID: The thread ID.

TOI: The thread of interest. This is the primary thread that will be affected by a command.

TRIGGER SET: The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

TRIGGERS: The effect during execution when program operations cause an event to occur (such as, arriving at a breakpoint).

TVDSVR PROCESS: The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

UNDIVING: The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you dive on the **undive** icon in the upper right-hand corner of the window.

UPPER BOUND: The last element in the dimension of an array or the slice of an array.

USER INTERRUPT KEY: A keystroke used to interrupt commands, most commonly defined as **^C** (Ctrl-C).

VARIABLE WINDOW: A TotalView window displaying the name, address, data type, and value of a particular variable.

VISUALIZER PROCESS: A process that works with TotalView in a separate window, allowing you to see a graphical representation of program array data.

WATCHPOINT: An action point specifying that execution should stop whenever the value of a particular variable is updated.

Citations

LLNL: This definition was taken from documention residing on the web site of the Lawrence Livermore National Laboratories. www.llnl.gov

:

Index

– difference operator 63

Symbols

separator character 80
$newval variable in watchpoints 226
$oldval variable in watchpoints 226
& intersection operator 63
. (dot) current set indicator 41
.tvd files 260
.Xdefaults file 72
/ slash in group specifier 46
< first thread indicator 41
= symbol for PC of current buried stack frame 173
> symbol for PC 173
@ symbol for action point 173
| union operator 63

A

A Slice of the Reassembled Array figure 109
-a switch 77
a width specifier 48
  examples 51
  general discussion 50
abbreviating commands 82
ac, see dactions command
action point identifiers 86, 129, 156
  never reused in a session 86
action points 86
  default for newly created 204
  deleting 123, 150, 242, 248, 250, 254, 261, 263
  disabling 152
  displaying 129
  identifiers 130
  information about 129
  loading saved information 130
  reenabling 156
  saving information about 130
  sharing 204
actionpoint command 123
actionpoint properties 123
actions, see dactions
adding a prototype 94
adding debugger information 67
adding group members 163
adding groups 162
adding members to a group 45
address callback 92
address procedure 103
address property 123
address, returning 103
address_callback property 254
addressing expressions 114
  format 115
advancing and holding processes 85
advancing by steps 211
advancing program execution 85
alias command 126
aliases
  built-in 82
  default 126
  group 82
  group, limitations 83
  removing 266
all width specifier 42
ambiguous scoping 81
appending to state variable lists 171
apply_prototype_to_focus routine 99
applying prototypes 98
architecture 205
arena specifiers 41
  defined 41
  incomplete 57
  inconsistent widths 58
arenas 157, 180

defined 39
iterating over 41
ARGS variable 77, 198, 201
modifying 78
ARGS_DEFAULT variable 77, 198, 201
clearing 78
arguments
command line 198
default 201
replacing 78
array prototypes 92
defining bounds 93
defining rank 93
distribution 93
array slice
initializing 9
printing 9
arrays
distributed 107
in global index space 108
run-time bounds 89
arriving at barrier 138
as, see dassign command
assembler instructions, stepping 215
assign, see dassign command
assigning output 76
assigning output to variable 76
assigning p/t set to variable 44
asynchronous processing 16
at, see dattach command
attach, see dattach command
attaching to parallel processes 134

B

b, see dbreak command
ba, see dbarrier
barrier
arriving 138
what else is stopped 137
barrier breakpoint 137, 139
barrier is satisfied 201
barrier points 37, 201
defined 86
barrier, see dbarrier
BARRIER_STOP_ALL variable 201
BARRIER_STOP_WHEN_DONE variable 201
base class 101
block, specifying in scope 81
blocking command input 224
blocking input 224
blocks, naming 80
bounds callback 93
bounds, returning 102
bounds_callback property 254
break, see dbreak command
breakpoint operator 64
breakpoints
barrier 137
default file in which set 143
defined 86, 137, 142
setting 11
setting at beginning of procedure 143
stopping all processes at 142
temporary 219
triggering 143
built-in alias 82
buried stack frame 172

C

C control group specifier 47
C language escape characters 132
C width specifier 48
call stack 222
displaying 233
callbacks
address 92
bounds 93
distributed 110
distribution 93
rank 93
structure 97
type 92
typedef 91, 93, 104
validation 91
capture command 76, 77, 128, 244
CGROUP 53
CGROUP variable 201
changing dynamic context 222
changing focus 157
changing process thread set 38
changing program state 69
changing state variables 200
changing value of program variable 132, 146, 165, 169, 183, 196, 217, 219
checkpoint, see dcheckpoint
checkpointing
preserving IDs 146
process state 145
reattaching to parallel 145
restarting 195
scope 145
socket issue 146
clarifying scope with dwhat 81
CLI
and Tcl 1, 67, 68
command results 69
components 67
defined 1
how it operates 67
initialization 71, 72
initialization file 71
interface 69
invoking program from shell example 73
not a library 68
output 76
relationship to TotalView 68
-s switch 71
scoping interpretation 79
starting 70
starting from command prompt 70
starting program using 73
using within Tcl, no differences 68
CLI commands
abbreviating 82
action points 121
alias 126
aliases 277, 281
assigning output to variable 76
capture 76, 77, 128, 244
dactions 129

dassign 132
dattach 85, 134
dbarrier 137
dbreak 142
dcheckpoint 145
dcont 148
ddelete 150
ddetach 151
ddisable 152
ddown 153
dec2hex 155
default focus 39
denable 156
dfocus 157
dgo 59, 160
dgroups 162
dhalt 168
dhold 169
dkill 74, 85, 170
dlappend 171
dlist 172, 204
dload 73, 74, 85, 175
dnext 177
dnexti 180
dout 183
dprint 186
dptsets 190
drerun 74, 193
drestart 195
drun 73, 78, 170, 197
drun, reissuing 198
dset 78, 82, 200
dstatus 209
dstep 41, 43, 59, 211
dstepi 215
dunhold 217
dunset 78, 218
duntil 219
dup 222
dwait 224
dwatch 225
dwhat 81, 229
dwhere 59, 233
dworker 235
environment 119
execution control 121
exit 238
focus of 277, 281
help 244
initialization 120
overview 119
program information 120
propagate_prototypes routines 98
prototype 89
quit 257
responding to 258
stty 259
summary 267
termination 120
TV::actionpoint 123
TV::errorCodes 236
TV::focus_groups 239
TV::focus_processes 240
TV::focus_threads 241
TV::group 242
TV::hex2dec 246
TV::image 94, 247
TV::image add 98
TV::process 94, 250
TV::prototype 94, 97
TV::prototype command 253
TV::respond 258
TV::source_process_startup 260
TV::thread 261
TV::type 95, 263
unalias 266
CLI prompt 73
CLI variables, see state variables
closed loop, see closed loop
clusterid property 250
co, see dcont command
code, displaying 172
command aliases 277, 281
command arguments 77
clearing example 78
passing defaults 78
setting 77
command focus 277, 281
command input, blocking 224
command line arguments 74, 198
Command Line command 70
Command Line Interpreter, see CLI
command output 128
command prompts 81
default 81
format 81
setting 82
starting the CLI from 70
command summary 267
COMMAND_EDITING variable 202
commands
interrupting 69
TV::image 94
user-defined 126
commands, responding to 258
compiler
adding debugging information 67
compiler information, interpreting 79
completion rules for arena specifiers 57
components of an executing program 1
conditional watchpoints 225
cont, see dcont command
continuation_sig property 261
continuous execution 69
control group 24
defined 23
overview 44
control group specifier 47
control in parallel environments 85
control in serial environments 85
control list element 203
Control-C 69
controlling program execution 85
count property 242
creating a group 162, 165
creating a struct transformation 95
creating commands 126
creating groups 28
creating new process objects 175
creating new processes 74

creating prototypes 253
creating the prototype 97, 106
creating threads 18, 160
critical regions 63
ctrl-d to exit CLI 238, 257
current list location 153
current set indicator 41
cyclic_array.tcl typemappings 281

D

D control group specifier 47
d, see ddown command
da_address procedure 113
da_distribution procedure 112
da_validate function 110
dactions command 129
daemons 16, 18
dassign command 132
datatype incompatibilities 132
dattach command 85, 134
dbarrier command 137
dbreak command 142
dcheckpoint command 145
  preserving IDs 146
  process 145
  reattaching to parallel 145
  scope 145
  socket issue 146
dcont command 148
ddelete command 150
ddetach command 151
ddisable command 152
ddown command 153
de, see ddelete command
deadlocks at barriers 139
debugger
  how it operates 67
  separate from program 67
debugger initialization 71, 72
debugger initialization file 71
  see also initialization
debugger PID 84
debugging option 67
debugging session 85
  ending 238
debugging techniques 35
dec2hex command 155
default aliases 126
default arguments 198, 201
  modifying 198
default control group specifier 47
default focus 54, 157
default process/thread set 38
default value of variables, restoring 218
default width specifier 42
defining prototypes 91
defining the current focus 204
delete, see ddelete command
deleting action points 123, 150, 242, 248, 250, 254, 261, 263
deleting groups 162, 164
deleting state variables 200
denable command 156
dfocus command 38, 157
  as modifier 39
  example 39
dgo command 59, 160
dgroups command 162
  –add 53, 163
  –add command 45
  –delete 164
  –intersect 164
  –list 164
  –new 165
  –remove 36, 165
dhalt command 168
dhold command 169
di, see ddisable command
difference operator 63
directory search paths 202
disable, see ddisable command
disabling action points 152
discarding buffered output 77
disconnected processing 16
display an object using a prototype 91
display call stack 233
displaying code 172
displaying current execution location 233
displaying error message information 205
displaying expressions 186
displaying help information 244
displaying information on a name 229
displaying values 186
distributed arrays 107
distribution callback 93, 110
  da_distribution 112
distribution_callback property 255
dividing work up 17
dkill command 74, 85, 170
dlappend command 171
dlist command 172, 204
dload command 73, 74, 85, 175
  returning process ID 76
dlopen() 94
dnext command 177
dnexti command 180
dout command 183
down, see ddown command
dpid 84, 201
dpid property 261
dprint
  parsing output 105
dprint command 186
dptsets command 190
drerun command 74, 193
drestart
  attaching automatically 195
  attaching to processes 195
  process state 195
drestart command 195
drun
  poe issues 199
drun command 73, 78, 170, 197
  reissuing 198
dset command 78, 82, 200
  creating a new variable 200
dstatus command 209
dstep command 39, 41, 43, 59, 211
dstepi command 215
duid property 250, 261
dunhold command 217

dunset command 78, 218
duntil
group operations 220
duntil command 60, 219
dup command 222
dwait command 224
dwatch command 225
dwhat command 81, 229
clarifying scope 81
dwhere command 59, 233
dworker command 235

E

effects of parallelism on debugger behavior 83
eliminating tab processing 173
en, see denable command
enable, see denable command
enabled property 123
ending debugging session 238
enum_values property 263
error message information 205
error operators 64
ERROR state 205
errorCodes command 236
errorCodes variable 236
errors, raising 236
escape characters 132
evaluating state 86
evaluation points
defined 86
see also dbreak
setting 11
executable property 251
executable, specifying name in scope 80
EXECUTABLE_PATH variable 135, 173, 202
setting 7
executing a start-up file 71
executing as one instruction 180
executing as one statement 177
executing assembler instructions 215
executing program, components 1
executing source lines 211
execution
controlling 85
halting 168
execution location, displaying 233
execve() 44
existent operator 64
exit command 238
expression arguments 79
expression evaluation 79
expression property 123
expression values, printing 186
expressions 63
extract_offset utility procedure 106

F

f, see dfocus command
figures
A Slice of the Reassembled Array 109
Five Processes and Their Groups on Two Computers 27
Five Processors and Processor Groups 25, 26
Four Processor Computer Networks 20
Four-Processor Computer 19
Internal View of std::vector 90
Mail with Daemon 16
Mapped std::vector<int> 88
Program and Daemons 16
Reassembled Display 108
Standard Display of struct "a" as an Array 108
Threads 21
Two Computers Working on One Problem 17
Uniprocessor 15
Unmapped std::vector <int> 88
User Threads and Service Threads 22
User, Service, and Manager Threads 23
Visualization of the Reassembled Array 109
Width Specifiers 43
file for start up 71
first thread indicator of < 41
Five Processes and Their Groups on Two Computers figure 27
Five Processors and Processor Groups figure 25, 26
focus
commands 277
default 157
defining 204
pushing 39
restoring 39
see also dfocus command
focus of commands 281
focus_groups command 239
focus_processes 240
focus_threads 241
fork() 44
fork_loop.tvd example program 72
four linked processors 19
Four Processor Computer Networks figure 20
Four-Processor Computer figure 19

G

-g option 67
g width specifier 48, 54
g, see dgo command
go, see dgo command
goal breakpoint 212
GOI, defined 40
gr, see dgroups
group aliases 82
limitations 83
group command 242
group members, stopping flag 204
group name 46
group number 46
group of interest 212
group stepping 60

group syntax 46
group number 46
naming names 46
predefined groups 46
GROUP variable 202
group width specifier 42
group width stepping behavior 212
group_indicator
defined 46
groups
accessing properties 242
adding 162
adding members 163
creating 28, 162, 165
defined 23, 24, 44
deleting 162, 164
intersecting 162, 164
listing 162, 164
modifying 202
naming 163
overview 23
placing processes in 135
relationships 43
removing 165
removing members 162
returning list of 239
setting 52
setting properties 242
GROUPS variable 203
groups, see dgroups

H

h, see dhalt command
halt, see dhalt command
halting execution 168
held operator 64
held property 251, 262
help command 244
hex2dec command 246
hexadecimal conversion 155
hold, see dhold
holding and advancing processes 85
holding processes 169
holding threads 138, 169
hostname property 251
how a debugger operates 67

I

I/O redirection 193, 197
id property 123, 242, 248, 251, 255, 262, 263
image add prototype command 94
image command 94, 98, 247
image, defined 94
image_id property 264
image_ids property 251
image_load_callbacks list 98, 99
images
getting properties 248
loading 99
setting properties 248
implicitly defined process/thread set 38
incomplete arena specifier 57
inconsistent widths 58
infinite loop, see loop, infinite
INFO state 205
information on a name 229
initial process 83
initialization file 71, 266
typical contents 71
initialization search paths 71
initializing an array slice 9
initializing an object's properties 97
initializing debugging state 71
initializing the CLI 71, 72
initializing TotalView after loading an image 99
input, blocking 224
inserting working threads 235
instantiating a prototype 88
instructions, stepping 215
interactive CLI 67
interface to CLI 69
interleaving messages 75
Internal View of std::vector figure 90
interpreting compiler information 79
interrupting commands 69
intersecting groups 162, 164
intersection operator 63
invoking CLI program from shell example 73
is_dll property 248
iterating over a list 58
iterating over arenas 41

K

k, see dkill command
kill, see dkill command

L

L lockstep group specifier 47, 48
l, see dlist command
Laminate command 107
language property 124, 255, 264
launching processes 197
LD_LIBRARY_PATH 72
length property 124, 264
levels, moving down 153
line numbers for specifying blocks 80
line property 124
lines for listing 204
LINES_PER_SCREEN variable 77, 203
list location 153
list, see dlist command
list_type 96
list_validate procedure 95
listing groups 162, 164
using a regular expression 164
lists with inconsistent widths 58
lists, iterating over 58
LM_LICENSE_FILE 72
lo, see dload command
load, see dload command
loading action point information 130
loading tvd files 260
lockstep group 25, 40, 63
defined 24
number of 45
overview 45
lockstep group specifier 47
lockstep list element 203

lookup_keys property 248
loop, infinite, see infinite loop

M

machine instructions, stepping 215
Mail with Daemons figure 16
make_actions.tcl sample macro 11, 72
manager property 262
manager threads 21, 26
manager threads, running 211
mandel.c 107, 110, 281
Mandelbrot set 107
Mapped std::vector<int> figure 88
matching processes 60
MAX_LIST variable 172, 204
member_type property 242
member_type_values property 243
members property 243
message interleaving 75
missing TID 44
mixing arena specifiers 58
modifying groups 202
more processing 77, 186
more prompt 77, 203, 244
mpirun 20
MPMD (Multiple Program Multiple Data) 2
multiple executables 2
multiprocess programs
  attaching to processes 135
  process groups 44
multiprocessing 19

N

n, see dnext command
name property 248, 255, 264
name, information about 229
names of symbols 78
namespaces 78, 200
  TV:: 78, 200
  TV::GUI:: 78, 200
nested subroutines
  stepping out of 183
new groups 165
newval variable in watchpoints 226
next, see command
nexti, see dnexti command
ni, see dnexti command
nodeid property 251
nonexistent operators 64
non-sequential program execution 69
nonstack opcodes 116

O

object
  displaying with a prototype 91
objects
  generating address 92
  initializing properties 97
oldval variable in watchpoints 226
omitting components in creating scope 81
omitting period in specifier 58
omitting width specifier 57, 58
opcodes
  nonstack 116
  special 117
  without opcodes 117
operands
  top of stack 116
operands without opcodes 117
operators
  - difference 63
  & intersection 63
  | union 63
  breakpoint 64
  error 64
  existent 64
  held 64
  nonexistent 64
  running 64
  stopped 64
  unheld 64
  watchpoint 64
out, see dout
output
  assigning output to variable 76
  discarding 77
  from CLI 76
  only last command executed returned 76
  printing 76
  returning 76
  when not displayed 76

P

p width specifier 48
p, see dprint command
p.t notation 41
p/t expressions 190
p/t set expressions 63
p/t sets
  arguments to Tcl 38
  defined 38
  grouping 64
  set of arenas 41
  syntax 42
p/t syntax
  group syntax 46
parallel environments
  execution control 85
parallel program, defined 83
parsing comments example 11
parsing dprint output 105
passing default arguments 78
pc property 262
pid specifier, omitting 57
piling up 62
POI, defined 40
preserving IDs in checkpoint 146
print, see dprint command
printing an array slice 9
printing expression values 186
printing information about current state 209
printing variable values 186
procedure, specifying name in scope 81
Process > Startup 160
process barrier 139
process barrier breakpoint, see barrier breakpoint

process command 94, 250
process group behavior 52
process group stepping 60
process group, see also control group
process groups 24, 44
synchronizing 61
process groups, see also groups
process information
saving 146
process list element 203
process numbers are unique 83
process objects, creating new 175
process stepping 59
process width specifier 42
omitting 58
process width stepping behavior 212
process/set threads
saving 44
process/thread identifier 83
process/thread notation 83
process/thread sets 84
as arguments 38
changing 157
changing focus 38
default 38
examples 44
implicitly defined 38
inconsistent widths 58
structure of 42
target 38
widths inconsistent 58
process_id.thread_id 41
process_load_callbacks 99
processes
and threads 1
attaching to 134, 175
creating new 74
current status 209
destroyed when exiting CLI 238, 257
holding 169
initial 83
properties 250
releasing 217
releasing control 151
restarting 193, 197, 272
returning list of 240
spawned 83
starting 193, 197, 272
stepping 59
synchronizing 61, 86
terminating 74, 170
when stopped 60
processors and threads 19
Program and Daemons figure 16
program control groups, defined 44
program control groups, placing processes in 135
program execution
advancing 85
controlling 85
program state
changing 69
program stepping 211
program symbols 79
program variable, changing value 132, 146, 165, 169, 183, 196, 217, 219
prompt
and width specifier 50
PROMPT variable 82, 204
prompting when screen is full 186
prototype command 89, 253
TV::prototype 90
prototype commands 106
prototype create command 97
prototype IDs 94
prototype properties 253
prototype property 264
prototype set command 94
prototypes 87
applied to image 248
applying 98
creating 97, 106, 253
defining 91
instantiating 88
language 91
properties 91, 107
regular expression 91
validating 91
validation callback 91
prototypes property 248
pthread ID 84
PTSET variable 204
ptsets, see dptsets
pushing focus 39

Q

quit command 257
quotation marks 132

R

r, see drun command
raising errors 236
rank callback 93
rank property 264
rank, returning 102
rank_callback property 255
read_store utility procedure 105
Reassembled Display figure 108
redefining the type 96
reenabling action points 156
registers, using in evaluations 143
regular expressions 89
in prototype 91
releasing control 151
releasing processes and threads 138, 217
removing aliases 266
removing group member 162
removing groups 165
removing worker threads 235
replacing default arguments 78
replacing tabs with spaces 204
replicated elements 114
rerun, see rerun command
respond 258
restart, see drestart
restarting processes 193, 197, 272
restarting program execution 74
restoring focus 39
restoring variables to default values 218
results of entering a CLI command 69
results, assigning output to variables 76

resuming execution 143, 148, 160, 170
returning error information 236
returning the address 103
returning the rank 102
returning the type 102
root path 205
Root window, starting CLI from 70
routines, stepping out of 183
rr, see drerun command
rules for scoping 80
run, see drun command
running operator 64
running through 62
running to an address 219
run-time bounds 89

S

S share group specifier 47
-s switch to CLI 71
S width specifier 48
s, see dstep command
sample programs
  make_actions.tcl 72
sane command argument 70
satisfaction set 138, 201
satisfaction_group property 124
saving action point information 130
saving process information 146
scope 79
scope of symbols 78
scoping as a tree 80
scoping rules 80
scoping, ambiguous 81
scoping, omitting components 81
screen size 203
scrolling output 77
search paths 202
search paths for initialization 71
separate semantics 67
server on each processor 17
service threads 21, 26
  waiting for 22
set expressions 63
set indicator, uses dot 41
set, see dset command
setting breakpoints 11
setting groups 52
setting lines between more prompts 203
setting terminal properties 259
SGROUP variable 204
share group
  defined 24
  overview 45
share group specifier 47
share groups 24, 45, 204
share list element 203
share property 124
SHARE_ACTION_POINT variable 204
share_in_group flag 204
shared library, specifying name in scope 80
shell, example of invoking CLI program 73
SHLIB_PATH 72
showing current status 209
si, see dstepi command
SILENT state 205
slash in group specifier 46
source code, displaying 172
source file, specifying name in scope 81
source_process_startup command 99, 260
sourcing tvd files 260
spawned process 83
special opcodes 117
specifier combinations 48
specifiers
  and dfocus 50
  and prompt changes 50
  examples 48
specifying groups 46
splitting up work 17
SPMD (Single Program Multiple Data) 2
stack and processes 1
stack frame 172
  moving down through 153
stack movements 222
Standard Display of struct "a" as an Array figure 108
starting a process 193, 197, 272
starting program under CLI control 73
starting the CLI 70
Startup command 160
start-up file 71
  tvdinit.tvd 126
state property 251, 262
state variables
  ARGS 77, 198, 201
  ARGS, modifying 78
  ARGS_DEFAULT 77, 198, 201
  ARGS_DEFAULT, clearing 78
  BARRIER_STOP_ALL 201
  BARRIER_STOP_WHEN_DONE 201
  CGROUP 201
  changing 200
  COMMAND_EDITING 202
  deleting 200
  EXECUTABLE_PATH 135, 173, 202
  GROUP 202
  GROUPS 203
  LINES_PER_SCREEN 77, 203
  MAX_LIST 172, 204
  PROMPT 82, 204
  PTSET 204
  SGROUP 204
  SHARE_ACTION_POINT 204
  STOP_ALL 204, 227
  TAB_WIDTH 173, 204
  THREADS 204
  TOTAL_VERSION 205
  TOTALVIEW_ROOT_PATH 205
  TOTALVIEW_TCLLIB_PATH 205
  VERBOSE 205
  viewing 200
  WGROUP 205
state, initializing 71
state_values property 251, 262
std::list definition 95
std::vector 88, 100

installing 90
stderr redirection 193, 197
stdin redirection 193, 197
stdout redirection 193, 197
step, see dstep command
stepi, see dstepi command
stepping 59
at process width 59
at thread width 59
default group 59
goals 60
group width behavior 212
machine instructions 180, 215
piling up 62
process group 60
process width behavior 212
running through 62
see also dnext command, dnexti command, dstep command, and dstepi command
target program 85
thread group 60
thread width behavior 211
threads 61
stepping a group 60
stepping a process 59
stepping a thread 59
stop group breakpoint 143
stop, defined in a multiprocess environment 85
STOP_ALL variable 201, 204, 227
stop_group flag 204
stop_when_done property 124
stop_when_hit property 124
stopped operator 64
stopped process
responding to resume commands 139
stopped thread 25
stopping execution 168
stopping group members flag 204
struct prototypes 92
struct transformation, creating 95
struct_fields property 264
structure callback 97
stty command 259
stty sane command 70
symbol lookup 80
and context 80
symbol names 78
specifying in scope 81
symbol scope 78
symbol scoping, defined 80
symbol specification, omitting components 81
symbols as arguments 79
symbols, interpreting 132
synchronizing processes 61, 86
syspid property 251
system PID 84
system TID 84
system variables, see state variables
systid 84
systid property 262

T

t width specifier 48
tab processing 173
TAB_WIDTH variable 173, 204
tabs, replacing with spaces 204
target process/thread set 38, 85
target processes 168
terminating 170
target program
defined 2
stepping 85
target property 264
Tcl
and CLI 67, 68
and the CLI 1
books for learning 2
CLI and thread lists 68
interpreter 1
version based upon 68
Tcl callback functions 87
terminal properties, setting 259
terminating debugging session 238
terminating processes 74, 170
thread barrier breakpoint, see barrier breakpoint
thread command 261
thread group behavior 52
thread group stepping 60
thread groups 24, 44
see also groups
thread ID 84
thread list element 203
thread numbers are unique 83
thread of interest 39, 41, 43, 211, 219
defined 41
thread stepping 61
platforms where allowed 59
thread width 59
thread width specifier 42
omitting 58
thread width stepping behavior 211
threadcount property 251
threads
and processes 1
creating 18, 160
current status 209
getting properties 261
holding 138, 169
identifying service 22
manager 21
not available on all systems 24
releasing 217
returning list of 241
service 21
setting properties 261
stepping 59
user 21
worker 23
workers 22
threads destroyed when exiting CLI 238, 257
Threads figure 21
threads model 18
threads property 251
THREADS variable 204
tid 84
TID missing in arena 44
TOI, defined 40
Tools > Command Line 70

top of stack operands 116
TotalView
executable 205
relationship to CLI 68
scoping interpretation 79
starting the CLI within 70
totalview command 71
TOTALVIEW_ROOT_PATH variable 205
TOTALVIEW_TCLLIB_PATH variable 205
TOTALVIEW_VERSION variable 205
totalviewcli command 70, 71, 73
triggering breakpoints 143
troubleshooting 4
TV:: namespace 78, 200
TV::actionpoint command 123
TV::errorCodes 236
TV::focus_groups command 239
TV::focus_processes 240
TV::focus_threads 241
TV::group 242
TV::GUI:: namespace 78, 200
TV::hex2dec 246
TV::image 94
TV::image add 98
TV::image add prototype 94
TV::image command 247
TV::image_load_callbacks list 98, 99
TV::process 94
TV::process command 250
TV::process_load_callbacks list 99
TV::propagate_prototypes 98
TV::prototype 253
TV::prototype command 89, 90, 106
TV::prototype create 97
TV::prototype set 94
TV::respond 258
TV::source_process_startup command 260
TV::source_process_startup routine 99
TV::thread command 261
TV::type 95, 114
TV::type command 263
tvd files 260
TVD.breakpoints file 130
tvdinit.tvd start-up file 126, 266
Two Computers Working on One Problem figure 17
type
returning 102
validating 110
type callback 92
type command 95, 114, 263
type property 124, 243, 255, 264
type transformation 87
as regular expression 89
casting to original 90
creating 89
defining type 92
how applied 90
parallel applications 107
redefining the type 96
underlying implementation 90
using 89
validating 95
type_callback property 256
type_values property 124, 243, 264
typedef callback 91, 93, 104
typedef_callback property 256

U

u, see dup command
ultimate base class 101
ultimate_base utility procedure 105
unalias command 266
unconditional watchpoints 225
unheld operator 64
unhold, see dunhold
union operator 63
Uniprocessor figure 15
unique process numbers 83
unique thread numbers 83
Unmapped std::vector <int> figure 88
unset, see dunset command
until, see duntil
up, see dup command
user created groups
modifying 202
user mode 21
user threads 21
User Threads and Service Threads figure 22
User, Service, and Manager Threads figure 23
user-defined commands 126
using quotation marks 132
utility procedures
extract_offset 106
read_store 105
ultimate_base 105

V

validate_callback property 256
validating the data type 110
validating the type 95
value for newly created action points 204
values, printing 186
variables
assigning command output to 128
assigning p/t set to 44
changing values 132, 146, 165, 169, 183, 196, 217, 219
printing 186
setting command output to 76
watched 226
watching 225
vector_address procedure 103
vector_bounds procedure 102
vector_rank procedure 102
vector_type procedure 102
vector_validate routine 100
VERBOSE variable 205
View > Laminate 107
viewing state variables 200
Visualization of the Reassembled Array figure 109

W

W width specifier 48

W workers group specifiers 47
wa, see dwatch command
wait, see dwait command
WARNING state 205
watch, see dwatch command
watchpoint operator 64
watchpoints 225
  $newval 226
  $oldval 226
  conditional 225
  defined 86
  information not saved 130
  length of 226
  supported systems 226
WGROUP 53
WGROUP variable 53, 205
wh, see dwhat command
what, see dwhat command
width specifier 41, 44
  omitting 57, 58
Width Specifiers figure 43
widths
  relationships 43
worker threads 23, 205
  inserting 235
  removing 235
worker, see dworker
workers group 25
  defined 24
  overview 45
workers group specifier 47
workers list element 203
working independently 17